阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

乘风者计划
THE WIND RIDERS PROGRAM

性能优化方法论

乘风者系列图书



明明如月

蚂蚁集团高级 Java 工程师
乘风者计划专家博主

性能优化的核心思想

性能优化在常见技术中的体现

技术和非技术层面优化双管齐下

性能优化，Java程序员学习与进阶的必经之路

VISIT…

LANZAROTE Caliente.com

扫码关注

明明如月

阿里云开发者“藏经阁”

海量电子书免费下载

目录

一、前言 **5**

1.1 关于本书 5

1.2 关于作者 5

二、性能优化的本质 **7**

2.1 性能优化的根本目的是什么? 7

2.2 出现性能问题的主要原因 7

2.3 性能优化的核心环节 8

2.4 寻找性能瓶颈 8

三、性能优化方法论的思想源泉 **10**

3.1 核心思想 10

3.1.1 开源和节流 10

3.1.2 堆“硬件”、升“软件” 10

3.1.3 权衡（trade-off） 11

3.2 具体来源 11

3.2.1 经典论断 12

3.2.2 百花齐放 13

四、性能优化的核心思想 **14**

4.1 增加资源 14

4.1.1 增加机器 14

4.1.2 升级配置 15

4.2 减少耗时操作 16

4.2.1 合并操作（化零为整） 16

4.2.2 压缩 21

4.2.3 复用 22

4.2.4 减少 IO 操作 23

4.2.5 减少上下文切换 26

4.2.6 减少操作指令 26

4.2.7 合理设置等待时间 28

4.3 提高资源利用率 28
4.3.1 空间换时间 28
4.3.2 同步转异步 38
4.3.3 串行转并行 39
4.3.4 降低冲突的范围 40
4.3.5 空间局部性 42
4.4 其他 42
4.4.1 提前处理 42
4.4.2 实时转离线 43
4.4.3 随机读写转顺序读写 43
4.4.4 就近原则 43
4.4.5 选择合适的数据结构和算法 44
4.4.6 加限制条件（技术层面） 44
4.4.7 CPU 密集型和 IO 密集型 45
4.4.8 根据技术特点去优化 45
4.4.9 全链路优化 46
4.4.10 多种手段相结合 46
4.5 产品层面 47
4.5.1 加限制条件（产品层面） 47
4.5.2 砍需求 48

五、注意事项 49
5.1 避免过早优化 49
5.2 考虑其他指标 49
5.3 优化体验 50

六、实际案例 51
6.1 案例描述 51
6.2 设计分析 51
6.3 案例总结 54

七、总结 55

一、前言

1.1 关于本书

本书主要分享自己在性能优化方面的一些思考。

性能优化是 Java 程序员学习和工作进阶过程难以绕开的一个重要话题。

很多人都想学好性能优化，希望能够在自己的工作中灵活运用，提升自己的技术水平，为用户提供良好的使用体验。

然而，很多人在工作中设计技术方案或者编码时缺乏系统的、方法论级别的理论指导，导致需要考虑性能优化的场景时，缺乏优化思路。

俗话说：“授人以鱼不如授人以渔”，本文不仅会讲性能优化有哪些具体的方法，还会讲解思想的来源。

本书会先讲述性能优化方法论的主要思想源泉，性能优化的本质；然后分别讲述性能优化方法论的核心方法，以及性能优化的注意事项等内容。讲解过程中会结合常见的 Java 中间件进行一些举例说明；最后会结合具体的案例，帮助大家理解性能优化方法论如何落地。

希望大家能够通过本书的学习，掌握性能优化的核心思路，帮助大家可以举一反三，可以从性能优化角度去学习 Java 中间件，去设计合理的性能优化技术方案。

1.2 关于作者

网名：明明如月，微信：mingmingruyuexz

蚂蚁集团高级 Java 工程师、阿里云开发者社区专家博主、 CSDN 博客专家。

博客总阅读量 350 万+，曾在慕课网出过“解锁大厂思维：剖析《阿里巴巴 Java 开发手册》”、“再学经典：《Effective Java》独家解析” 技术专栏，有多篇技术文章被知名技术公众号转载。

二、性能优化的本质

2.1 性能优化的根本目的是什么?

可能很多人没有认真思考过：“为什么我们需要进行性能优化？”这个问题。

在我看来，性能优化是为了”解决良好的用户体验和资源的有限性之间的矛盾”。

首先，我们性能优化一般都是追求更快的响应速度，通常最终目的是为了获得更好的用户体验。

这里所说的资源是指广义上的资源，“资源的有限性”包括：资金成本的有限性，人力资源的有限性，服务器等硬件资源的有限性，时间的有限性等。

如果资源是无限的，如开发周期很长，投入的人力很多，服务器资源充足，甚至服务器内存是无限大的，那么设计出来的产品可能就不需要花费太多精力去优化。

2.2 出现性能问题的主要原因

通常，产品发布早期由于用户量较少，不太容易出现性能问题或者通常不会太去关注性能问题；但随着业务的不断发展，性能问题就逐渐暴露出来。

导致性能问题的原因有很多，常见的原因有：

- 项目工期紧张，设计阶段技术方案考虑不充分；
- 项目中使用了不合理的数据结构或算法;
- 系统架构设计不合理;
- 同步执行耗时任务;
- …

2.3 性能优化的核心环节

不知道大家有没有深入思考过，我们平时写代码到底在写什么？

可能有些人会调侃地说：“无非就是 CURD”。

其实我们编码都是围绕着输入、处理和输出三个主要环节展开的。



因此，性能优化也要着重从这三点进行考虑。

如考虑如何更快地查询出数据，更快地对数据进行处理，更快地传输数据等。

通常来说，处理部分是其中最核心的环节，也是优化的重点。

2.4 寻找性能瓶颈

性能优化的前提是要找到性能瓶颈，通常需要通过自测、压测、耗时告警日志、数据库慢查询日志、调用链路追踪技术等手段，发现性能瓶颈。

通常在实际开发中，某个接口的响应时间过长影响用户体验，如果条件允许，就要考虑优化。

通常 2C 的业务更关注性能优化，2B 的业务相较于 2C 的业务通常来说性能问题只要不是太严重，通常没那么紧迫。

产生性能瓶颈的原因有很多，如使用的算法时间复杂度较高、数据库查询语句没有覆盖到索引、调用链路过长等。

如果需要分析接口找出性能瓶颈，推荐大家使用 arthas，可以使用其中的 trace 命令，该命令可以追踪调用链路，给出调用的耗时。

Arthas

如使用 trace 命令，对某个耗时较长的接口进行分析： trace com.xxx.service.impl.AServiceImpl refresh，给出下面结果：

```
Affect(class-cnt:2 , method-cnt:2) cost in 525 ms.
`---ts=2020-0X-0Y 13:33:18;thread_name=DubboServerHandler-127.0.0.1:20880-thread-
36;id=24e;is_daemon=true;priority=5;TCCL=com.mmm.WWWClassLoader@4362d7df
    `---[1761.834357ms] com.xxx.service.impl.AServiceImpl$$EnhancerBySpringCGLIB$$e3cd7543:refresh()
        +---[0.017066ms]
com.xxx.service.impl.AServiceImpl$$EnhancerBySpringCGLIB$$e3cd7543:$jacocoInit()
        `---[1761.00347ms] org.springframework.cglib.proxy.MethodInterceptor:intercept()
            `---[1757.647111ms] com.xxx.service.impl.AdServiceImpl:refresh()
                +---[0.006629ms] com.xxx.biz.yyy.service.impl.AServiceImpl:$jacocoInit()
                +---[0.004073ms] java.util.Collections:singletonList()
                +---[1709.203302ms] com.yyy.service.impl.AServiceImpl:refreshSomeThings()
                `---[48.135719ms] com.yzzzz.service.impl.AServiceImpl:createSurvey()
```

根据上面的结果可以看出，com.yyy.service.impl.AServiceImpl:refreshSomeThings 耗时最长，可以继续再 trace 耗时最多的子函数，最终定位到最影响耗时的函数上。

找到耗时最长的环节之后，根据具体代码推断出耗时长的原因，然后再针对性地进行优化。

优化后可以通过性能测试、压力测试等手段验证性能优化的有效性。

三、性能优化方法论的思想源泉

本小节介绍自己对性能优化方法论的思想来源。

3.1 核心思想

3.1.1 开源和节流

既然，性能问题是”良好的用户体验和有限的资源之间的矛盾” 导致的。

那么，我们如何解决这个矛盾呢？换句话说，资源不足怎么办？

我们可以将问题进行转化，换个问法：“如果你这个月钱不够花怎么办”？

可能很多人会说：想办法多挣点，比如多加点班（如果有加班费的话）、多干一些兼职、可以向朋友借点，可以省着点花。

这些方法的背后闪烁着两个重要思想：“开源和节流”！！

开源，即投入更多的资源。如增加更多服务器、提高服务器的配置，投入更多的开发人员等。

节流，即提高资源利用率，减少资源浪费。如采用时间复杂度更低的算法，通过压缩来降低存储和传输的数据量等。

3.1.2 堆“硬件”、升“软件”

从宏观上讲，计算机是由硬件和软件组成的。

那么，如何让一个计算机性能更好呢？

一般来说，需要：堆硬件、升软件。如提高硬件配置，对软件进行优化。

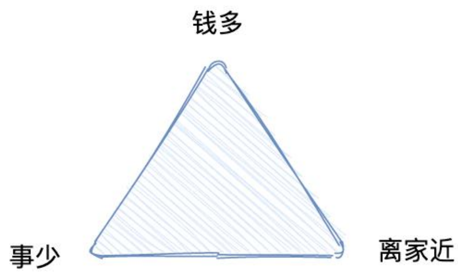
其实性能优化的主要方法也来源于此，性能优化的宏观思路就是“堆硬件，升软件”。

这里的硬件指机器的数量和机器的配置等；软件包括优化算法、架构等。

本质上和开源节流的思想是一致的。

3.1.3 权衡（trade-off)

大家找工作很难找到”钱多、事少、离家近”的工作。然而现实情况是，通常只能得其中一二。





性能优化很多时候也是一种权衡，在性能优化的路上，通常要做：用户体验和成本的权衡，投入产出比的权衡。

很多团队在产品研发初期人力资源有限，加上快速上线的压力，一般看重满足功能要求，只要不出现难以接受的性能问题即可。此外，项目也有优先级，可能另外一个项目优先级非常高，性能优化的事情可能就暂时延后。不管是通过”堆硬件” 还是通过”升软件” 的方式进行性能优化，都会面临着低延时和高成本的矛盾。

然而一般来说，用户体验和成本成反比，架构师或者开发人员要做的是，根据产品发展阶段和资源的情况去考虑性能优化的问题。

不管是从产品的发展阶段，还是团队内项目的优先级，还是技术角度上性能优化的投入和产出比来说，性能优化都不是简单地追求”最优化”，而是需要结合实际情况寻找优化的平衡点。

3.2 具体来源

前面讲到性能优化的核心思路是：开源和节流，堆硬件和升软件，对很多人来说太宏观。
那么如何寻找靠谱的具体的性能优化方法呢？

3.2.1 经典论断

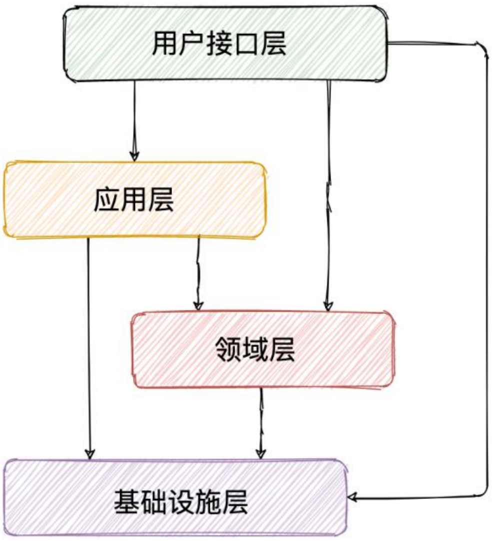
一些计算机科学、软件工程的一些经典论断，如 David Wheeler 的经典论断：

All problems in computer science can be solved by another level of indirection(the fundamental theorem of software engineering)

即计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决（软件工程的基础理论）



如在数据库和用户访问之间增加缓存层，可以极大加快访问速度；在写操作和磁盘中间增加内存缓冲区这个中间层，极大提高了 IO 的性能；再比如在应用程序和数据库之间，增加分库分表中间件作为中间层，突破了单库限制，提高了读写性能。如下图所示，在应用服务层和数据访问层中间增加领域层，可以更好地表达业务，更好地封装变化，实现更好高内聚、弱耦合的目标。

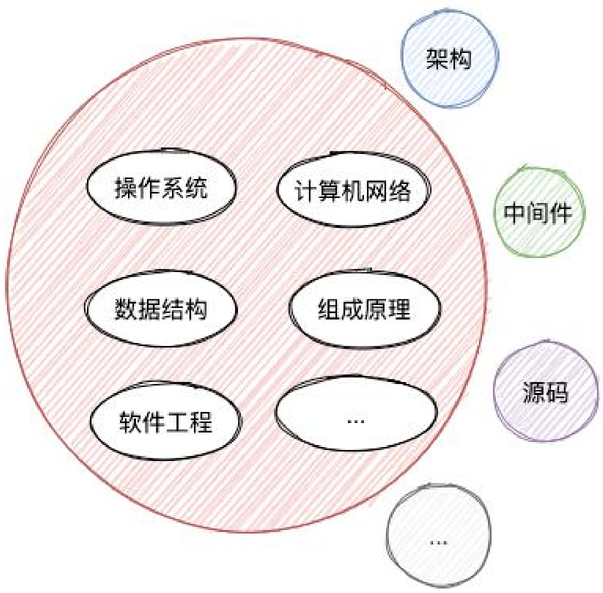


3.2.2 百花齐放

此外，专业基础是性能优化灵感的重要来源，如 CPU 的多级缓存思想、文件缓冲区的思想、进程调度算法等。

不同的数据结构都有自己的特点，都是为了解决某一类问题，适合某些场景，有其优缺点。选择合适的数据结构也可以实现性能优化。

不同的算法的时间和空间复杂度也各不相同，选择性能更优的算法也可以提高性能。



很多架构就是为了性能优化而设计的，如读写分离、分库分表、分布式架构；

很多中间件的设计中体现出诸多性能优化的思想，如 ES 的倒排索引、索引刷盘机制；

JDK 源码和很多优秀的开源项目也包含着很多性能优化的典型实践，如内存缓存、分段加锁、写时复制等。

四、性能优化的核心思想

首先，建议大家采用先猜想后验证的方式进行学习。即看到下面标题或者文章目录时，先猜想一下该标题下大概会包括哪些案例，然后和给出的条目进行对比，学习效果更好。

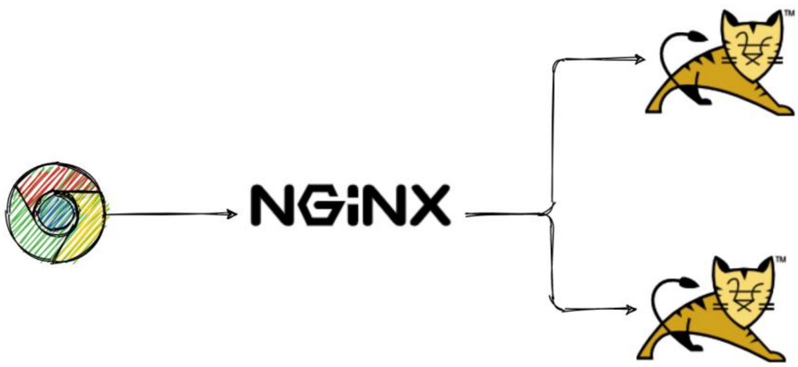
4.1 增加资源

4.1.1 增加机器

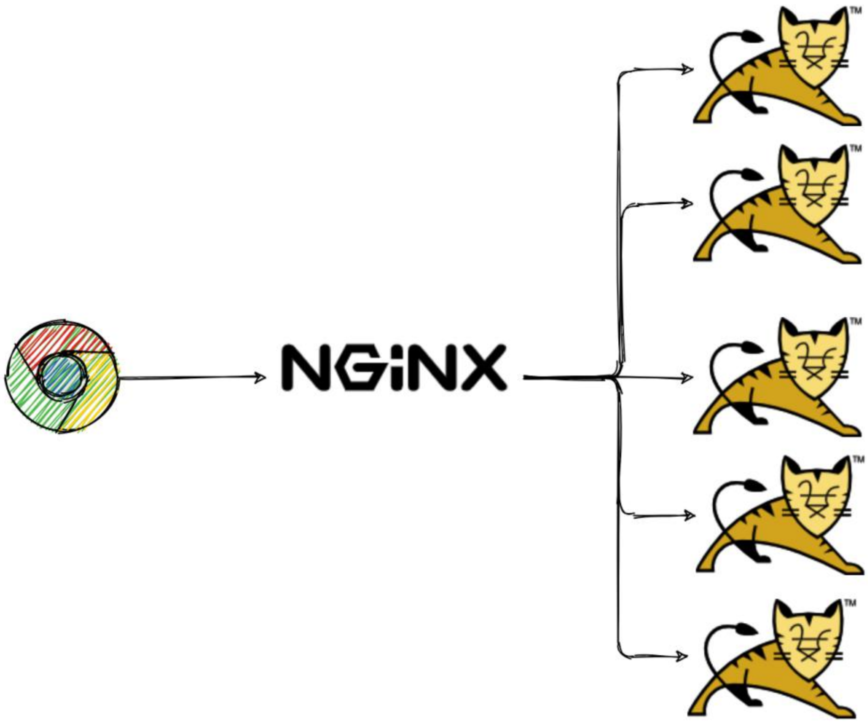
比如由单个 WEB 服务器来响应用户请求，改为通过 Nginx 等负载均衡工具将请求分发到多台服务器。



这就相当于原本店铺里只有一个服务员，一个服务员同时只能接待 5 位客人，现在又多招聘几个服务员，这样能够同时接待顾客数量会更多一些。



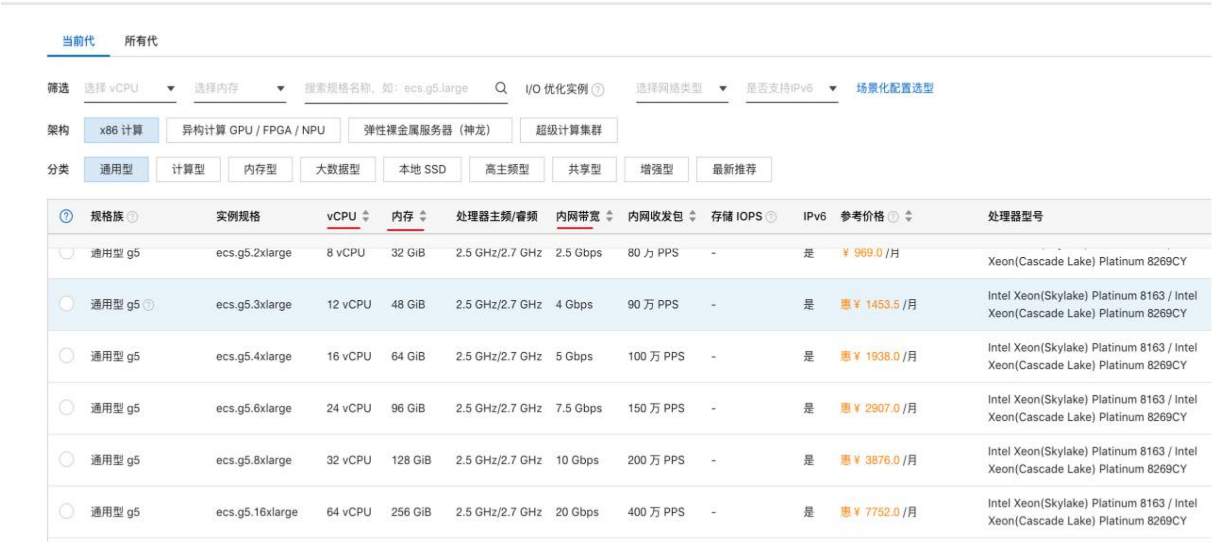
或者为集群添加更多机器。



通常很多站点都会在某些大型活动（如限时秒杀、微博热帖、高考查成绩、热门直播等）前或过程中，通过手动或者自动”扩容” 增加更多机器的方式来支撑更多流量。

4.1.2 升级配置

增加配置，通常是更换性能更好的 CPU、增加内存、增加宽带、使用固态硬盘、增加磁盘空间等。下图为阿里云购买服务器页面的截图，其中的选项就是配置相关的关键指标。



规格族	实例规格	vCPU	内存	处理器主频/睿频	内网带宽	内网收发包	存储 IOPS	IPv6	参考价格	处理器型号
通用型 g5	ecs.g5.2xlarge	8 vCPU	32 GiB	2.5 GHz/2.7 GHz	2.5 Gbps	80 万 PPS	-	是	¥ 969.0 /月	Xeon(Cascade Lake) Platinum 8269CY
通用型 g5	ecs.g5.3xlarge	12 vCPU	48 GiB	2.5 GHz/2.7 GHz	4 Gbps	90 万 PPS	-	是	惠 ¥ 1453.5 /月	Intel Xeon(Skylake) Platinum 8163 / Intel Xeon(Cascade Lake) Platinum 8269CY
通用型 g5	ecs.g5.4xlarge	16 vCPU	64 GiB	2.5 GHz/2.7 GHz	5 Gbps	100 万 PPS	-	是	惠 ¥ 1938.0 /月	Intel Xeon(Skylake) Platinum 8163 / Intel Xeon(Cascade Lake) Platinum 8269CY
通用型 g5	ecs.g5.6xlarge	24 vCPU	96 GiB	2.5 GHz/2.7 GHz	7.5 Gbps	150 万 PPS	-	是	惠 ¥ 2907.0 /月	Intel Xeon(Skylake) Platinum 8163 / Intel Xeon(Cascade Lake) Platinum 8269CY
通用型 g5	ecs.g5.8xlarge	32 vCPU	128 GiB	2.5 GHz/2.7 GHz	10 Gbps	200 万 PPS	-	是	惠 ¥ 3876.0 /月	Intel Xeon(Skylake) Platinum 8163 / Intel Xeon(Cascade Lake) Platinum 8269CY
通用型 g5	ecs.g5.16xlarge	64 vCPU	256 GiB	2.5 GHz/2.7 GHz	20 Gbps	400 万 PPS	-	是	惠 ¥ 7752.0 /月	Intel Xeon(Skylake) Platinum 8163 / Intel Xeon(Cascade Lake) Platinum 8269CY

4.2 减少耗时操作

要提高性能，通常需要减少操作的耗时。想要减少耗时，就要了解常见传输媒介的耗时情况。
下图为各种操作的时间量级参考表：

事件	延时	相对时间比例
1 个 CPU 周期	0.3 ns	1 s
L1 缓存访问	0.9 ns	3 s
L2 缓存访问	2.8 ns	9 s
L3 缓存访问	12.9 ns	43 s
主存访问（从 CPU 访问 DRAM）	120 ns	6 分
固态硬盘 I/O（闪存）	50–150 μs	2–6 天
旋转磁盘 I/O	1–10 ms	1–12 月
互联网：从旧金山到纽约	40 ms	4 年
互联网：从旧金山到英国	81 ms	8 年
互联网：从旧金山到澳大利亚	183 ms	19 年
TCP 包重传	1–3 s	105–317 年
OS 虚拟化系统重启	4 s	423 年
SCSI 命令超时	30 s	3 千年
硬件虚拟化系统重启	40 s	4 千年
物理系统重启	5 m	32 千年

(图片来源：《性能之巅》[1])

这张图是我们后续很多性能优化方法的主要依据。

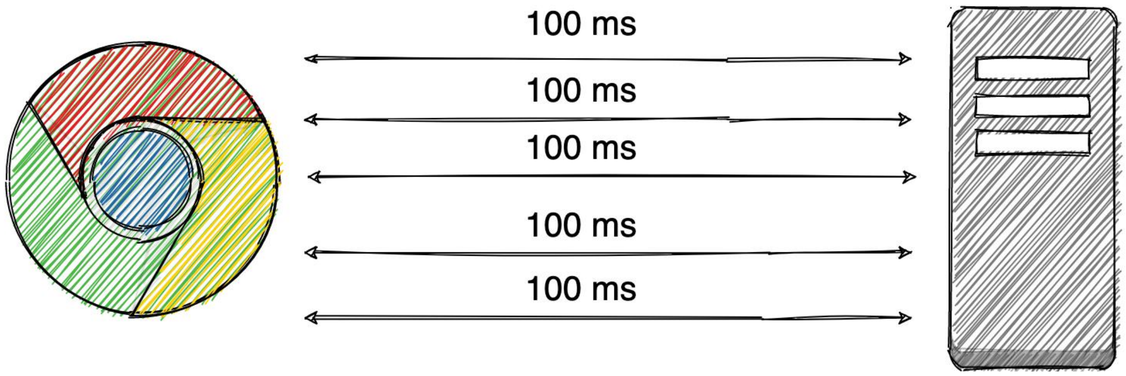
4.2.1 合并操作（化零为整）

合并操作是性能优化非常典型和重要的思想。

比如渲染某个前端页面，前端需要请求多个 js 文件，可以将多个 js 文件合并成一个，来减少向服务端发起请求的次数。

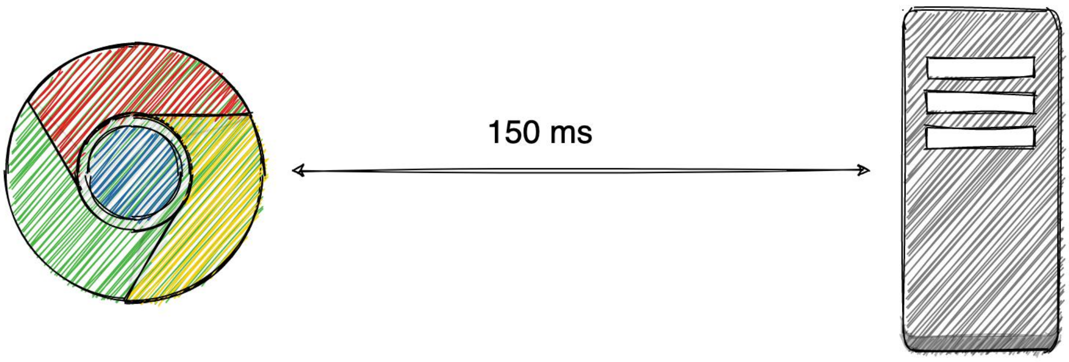
比如后端服务在某个请求中需要构造不同的请求，多次调用同一个二方接口，此时，可以使用批量查询接口，而不是 for 循环中执行单个请求再去处理。

如下图所示，某个功能，需要在循环中请求十几次甚至几十次二方或者三方接口：



假设每次耗时 100ms，请求 20 次，就是 2 秒钟。

如下图所示，可以通过调用批量接口，发起一次网络请求获取十几条数据或者几十条数据：



同样的功能，可能 100 ms 更多一点就搞定了。

很多中间件在设计时，也会提供一些批量接口。

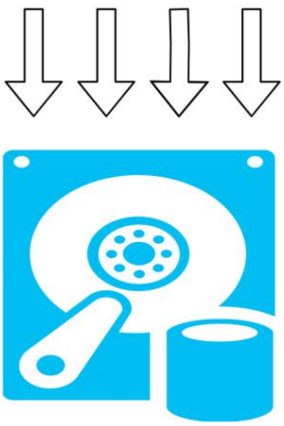
如 hbase-client 中就提供了批量查询接口：

org.apache.hadoop.hbase.client.Table#get(java.util.List<org.apache.hadoop.hbase.client.Get>)

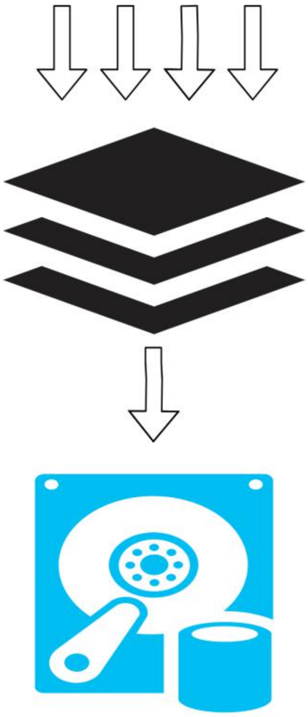
```java
/**
 * Extracts specified cells from the given rows, as a batch.
 *
 * @param gets The objects that specify what data to fetch and from which rows.
 * @return The data coming from the specified rows, if it exists.  If the row specified doesn't
 *   exist, the {@link Result} instance returned won't contain any
 *   {@link org.apache.hadoop.hbase.Cell}s, as indicated by {@link Result#isEmpty()}. If there
 *   are any failures even after retries, there will be a <code>null</code> in the results' array
 *   for  those Gets, AND an exception will be thrown. The ordering of the Result array
 *   corresponds to  the order of the list of passed in Gets.
 * @throws IOException if a remote or network exception occurs.
 * @since 0.90.0
 * @apiNote {@link #put(List)} runs pre-flight validations on the input list on client.
 *          Currently {@link #get(List)} doesn't run any validations on the client-side,
 *          currently there is no need, but this may change in the future. An
 *          {@link IllegalArgumentException} will be thrown in this case.
 */
default Result[] get(List<Get> gets) throws IOException {
    throw new NotImplementedException("Add an implementation!");
}
```

再如 Redis 中的 mget 命令，可以批量获取 key 的值；ES 中也提供了 mget 批量查询 api。

大家都知道 IO 操作通常和读写内存、CPU 缓存等相比非常耗时，如果想进行性能优化，就要考虑减少 IO 操作。



那么我们可以将多个写操作先写到内存缓冲区中，达到一定的条件再落盘。



Java 中也提供了如 java.nio.ByteBuffer 和 java.io.BufferedOutputStream 等。

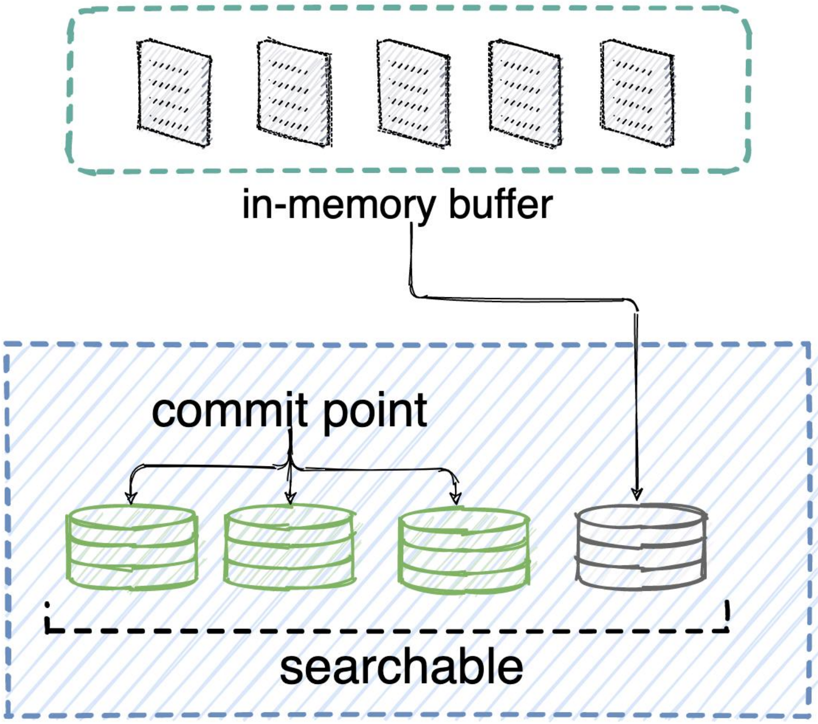
下面是 BufferedOutputStream 的源码注释：

```java
/**
 * The class implements a buffered output stream. By setting up such
 * an output stream, an application can write bytes to the underlying
 * output stream without necessarily causing a call to the underlying
 * system for each byte written.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */
public class BufferedOutputStream extends FilterOutputStream {
 // 省略
}
```

从注释中也可以看出，应用可以使用该类将字节写入到这里，而不是每个字节都调用底层写入方法，本质上就是合并请求。

MySQL 、Elasticsearch 等很多存储相关的功能都用到了”缓冲区”的思想。

以 Elasticsearch 为例，索引是映射类型的容器，一个 ES 索引是独立的一个大量文档的集合。每个索引存储在磁盘上的同组文件中，索引存储了所有映射类型的字段和设置。如下图所示，数据先存储到内存缓存功能区中，当 refresh 发生时，数据被提交到 segment 然后可被搜索，每个索引可以设置 refresh 间隔，可以通过 refresh_interval 参数控制。



由于 refresh 的耗时相对较长，可以通过控制刷盘的间隔来提高性能。而这个时间间隔的设置需要进行权衡，如果设置间隔太久，会导致新的数据需要很久才可被搜索到，影响用户体验；设置的时间太短，造成性能损耗。

“当一条数据需要更新时， InnoDB 引擎就会先把记录写到 redo log 里面，并更新内存，这个时候更新就算完成了。同时，InnoDB 引擎会在适当的时候，将这个操作记录更新到磁盘里面”[2]。也是通过合并的方式减少写盘次数进而优化性能。

HBase 中每次刷新 memstore 都会产生新的 HFile，由于 HFile 存储在磁盘上，就需要寻址操作。传统的硬盘寻址较慢， HFile 多了之后寻址操作就增多，效率就不高。为了防止 HFile 过多，同时为了减少碎片，HBase 会执行一些合并操作。

4.2.2 压缩

对要存储或传输的数据进行压缩，可以减少资源占用，提高传输速率。

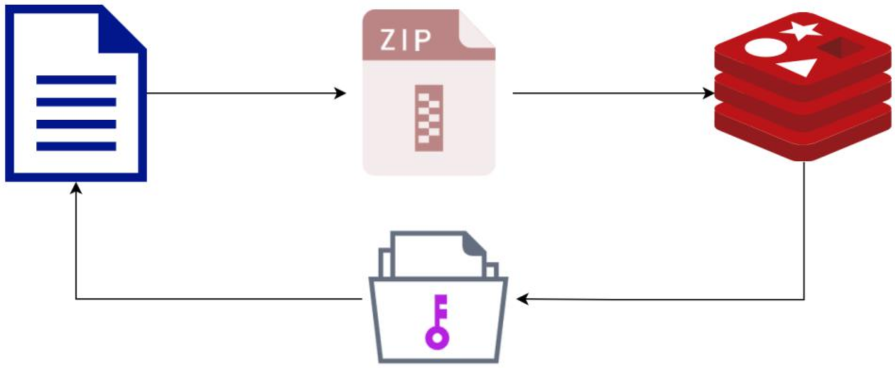
比如前端可以对 js 或者 css 文件进行压缩，来减少传输的数据量，加快资源加载速度。
也可以在使用资源时，默认对资源自动压缩。

如通过微信发送图片或者视频时，默认会自动压缩，必要时可以选择原图进行发送。



查看时只加载预览图，在必要时可以选择查看原图或者选择清晰度更高的视频。如 QQ 空间相册、爱奇艺/ B 站等视频的清晰度切换等。

比如后端可以对将要存储到 redis 中的大段文本数据进行压缩，然后再存储，使用前再解压。

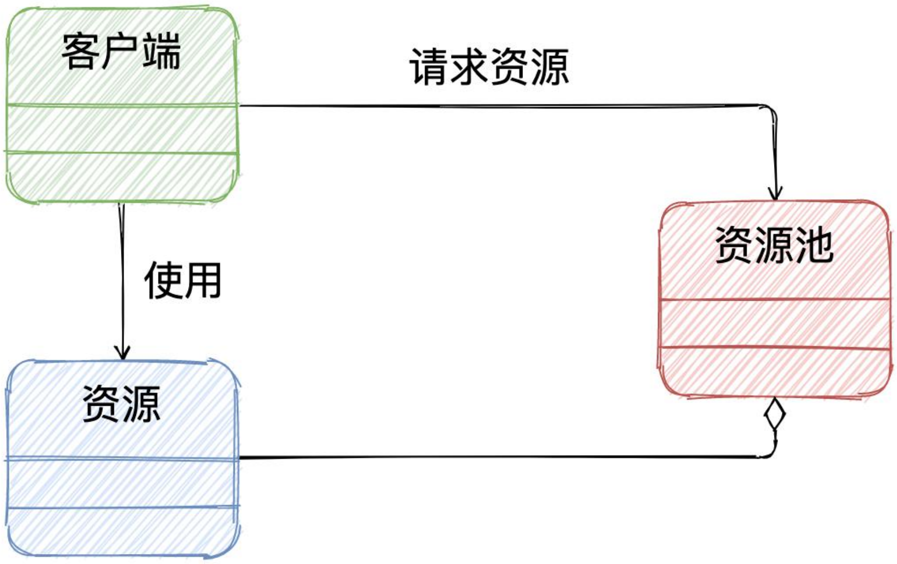


4.2.3 复用

复用是减少资源使用的非常经典的方式。包括 各种连接池、线程池在内的对象池模式是复用典型应用。

它们的核心思想都是重用和共享创建代价比较昂贵的对象。

其结构如下：



基本思想：当客户端需要资源时，向资源池发起申请，资源池检查后获取第一个可用资源给客户端使用；客户端使用后归还资源到资源池重复使用。

比如 Http 长连接就是在同一个连接中发起多次请求来提高性能的； 数据库连接池，也是通过复用连接来提高性能的模式。

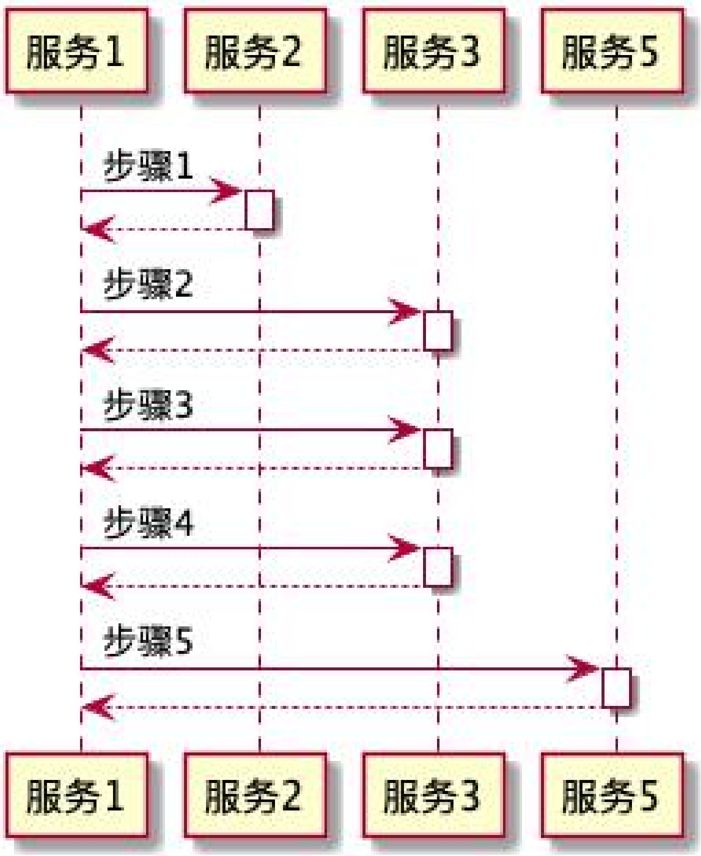
4.2.4 减少 IO 操作

下面只是给出几个实际中常见的具体案例，大家要根据实际情况进行举一反三。

减少不必要的调用

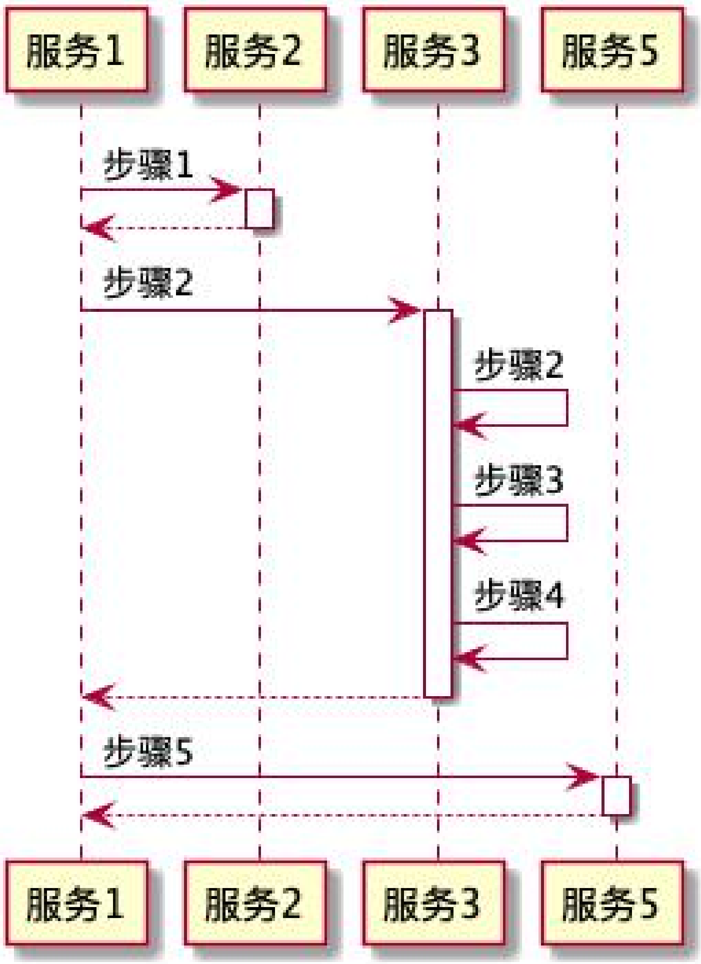
有时，前端由于架构设计不够合理等原因，会进行一些没有必要的同步调用，造成响应时间无辜被拖长。

此时，需要去掉不必要的调用（尤其是同步调用）。



做新的项目时，有时候需要协调多个下游，如果有些接口没有必要调用，就要减少调用。

如果有些接口应该是你的下游给封装起来的，下游的服务应该对其进行封装，避免造成上游进行不必要的接口调用。

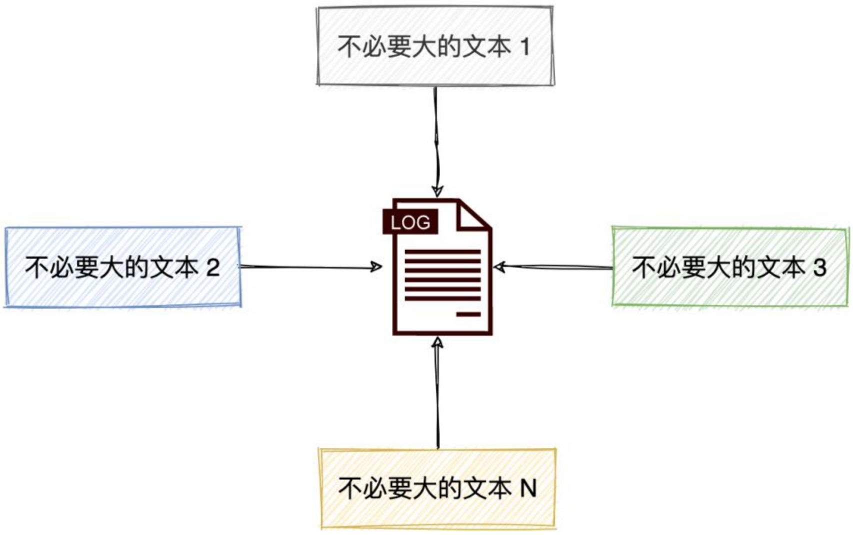


调用额外的接口增加耗时，而且很多场景下不符合迪米特法则。

注：迪米特法则又叫最小知道原则，一个类对其他类知道的越少越好，只和最直接的“朋友”对话（talk only to your immediate friends）。本质上还是为了实现高内聚、弱耦合。

减少不必要的日志输出

大家在编写代码时，可能会（同步）打印一些没有必要的大文本日志，当请求量较多时也会影响性能，影响系统响应时间或者浪费存储空间。

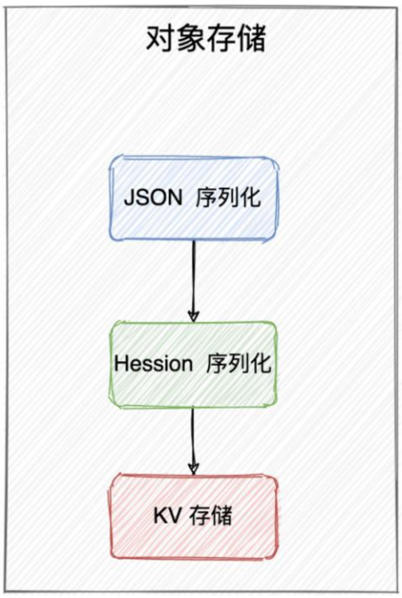


因此建议大家只打印必要的日志，对于大对象只打必要的字段。

减少不必要的转换

比如有时候需要将内存对象持久化到一些 KV 存储中，由于有些序列化方式需要实现序列化接口，而有些对象没有实现序列化接口从而不支持某种二进制序列化方式，有些人会选择先进行 JSON 序列化成字符串然后再进行存储，这样做性能很差。

如果 KV 存储要求实现序列化接口，如果想要序列化没有实现序列化接口的二方或者三方 jar 包中的类，可以定义一个具有相同属性的类，转换后再进行序列化。



4.2.5 减少上下文切换

频繁地上下文切换也会造成性能损耗。

有些场景下使用多线程执行，由于频繁地上下文切换造成性能损耗反而比使用单线程耗时更长。自旋锁是减少上下文切换的一个优化案例。

如果有一个以上处理器，能让两个或者以上的线程同时并行执行，可以让后面请求锁的线程“稍等一下”，不放弃处理器的执行时间，看看持有锁的线程是否很快就释放。如果自旋超过限定的次数仍然没有成功获取锁，就应当使用传统的方式挂起线程。

4.2.6 减少操作指令

大家在看一些框架源码时，可能会见到类似下面的写法：

```java
public class LocalDemo {

    private List<String> data = new ArrayList<>();

    public void someMethod(String param) {
        List<String> data = this.data;
        if (data != null && data.size() > 0 && data.contains(param)) {
            System.out.println(data.indexOf(param));
        }

    }

}
```

可能有些朋友会好奇，为啥要定义一个局部变量 data 呢？

为什么不像下面这么写呢？

```java
public class ThisDemo {

    private List<String> data = new ArrayList<>();

    public void someMethod(String param) {
        if (data != null && data.size() > 0 && data.contains(param)) {
            System.out.println(data.indexOf(param));
        }
    }
}
```

大家动手用 javap 进行反汇编之后你会发现，如果当前函数多次使用 data 时，第一种写法指令更少。

第一种写法，第一次将 data 对象存储到当前栈帧的局部变量表中，使用时直接从局部变量表中获取，而第二种写法需要先装载 this 然后通过 getfield 方法来获取属性。

想了解详细内容，可以参考我一篇专题文章：《为什么推荐大家学习 Java 字节码》

有些前置判断应该放在尽量靠前面的位置，避免做了一系列不必要的操作。

伪代码如下：

```java
public Result someMethod(User user){
    Result result = new Result();

    Data1 data1   =   xxxClient.querySome1();
    result.setData1(data1);

    Data2 data2   =    yyyClient.querySome2();
    result.setData2(data2);

    if(!user.hasAccess()){
        throw new IllegalArgumentException("没有权限");
    }

    Data3 data3   =    zzzClient.querySome3();
    result.setData3(data3);

    return result;
}
```

如果判断不成立，抛出异常，前面的查询操作都浪费了。

4.2.7 合理设置等待时间

执行远程接口调用时，要设置合理的等待时间。

如果等待的时间过长，那么即使能够获得数据，也很影响用户体验。与其这样，比如设置用户能够接受的超时时间，超时时让用户重试。

4.3 提高资源利用率

4.3.1 空间换时间

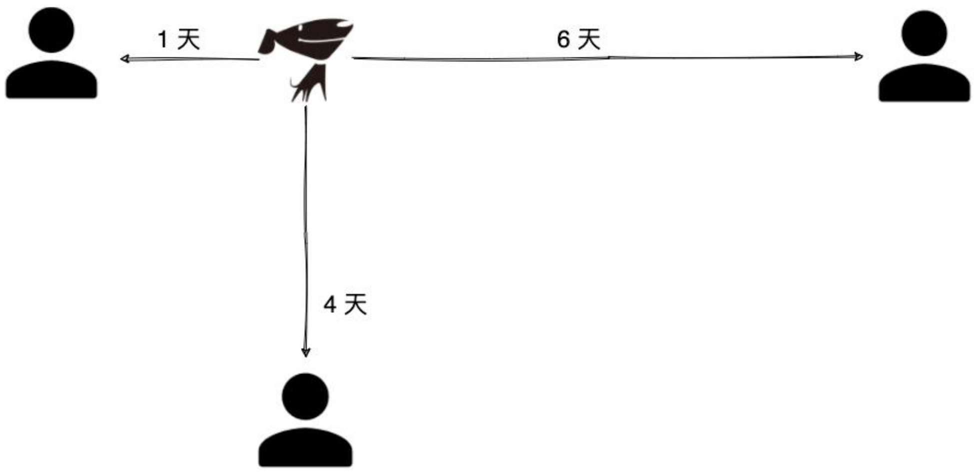
空间换时间是性能优化最常用的手段之一。

其中缓存就是空间换时间的一种典型应用。



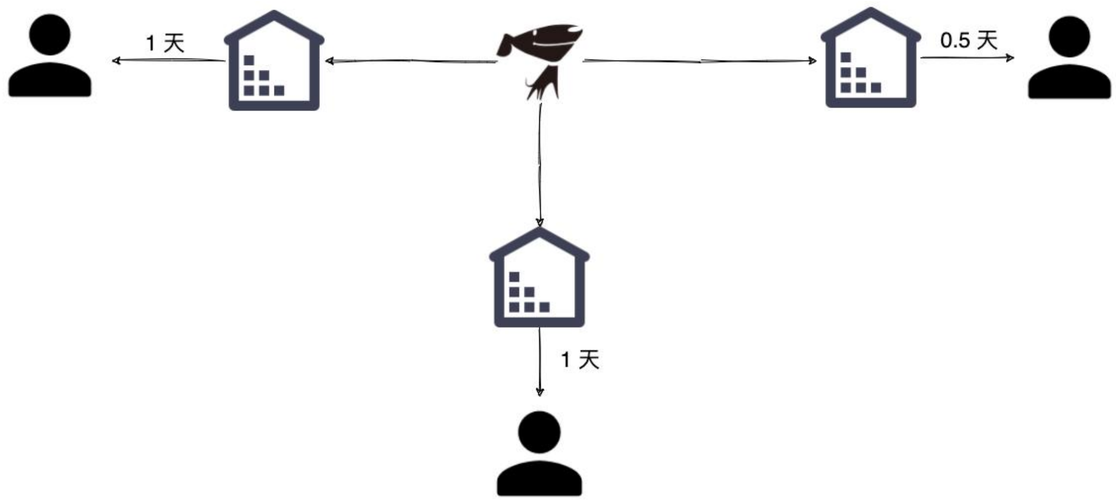
CPU 缓存、浏览器缓存、CDN 缓存、DNS 缓存、内存缓存、 Redis 缓存等，它们都是将数据缓存在离使用者更近的地方，或者读取速度更快的存储介质中，通过空间换时间的方式实现性能优化的。

在展开讲解之前，大家想想如果电商平台自营商品都从中央仓库发货，近的用户收货时间会短一些，远的用户收货时间可能会很长。

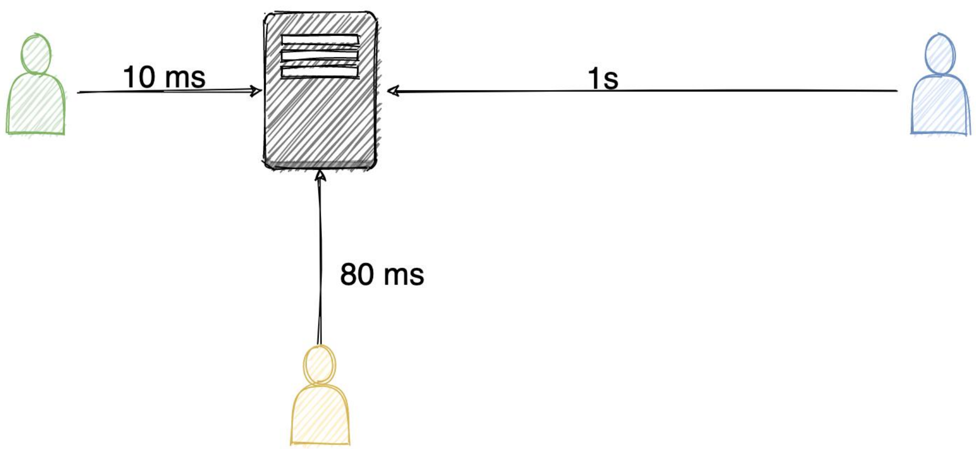


但是，很多朋友知道某东热门商品收货非常快，不知道大家是否知道原因。

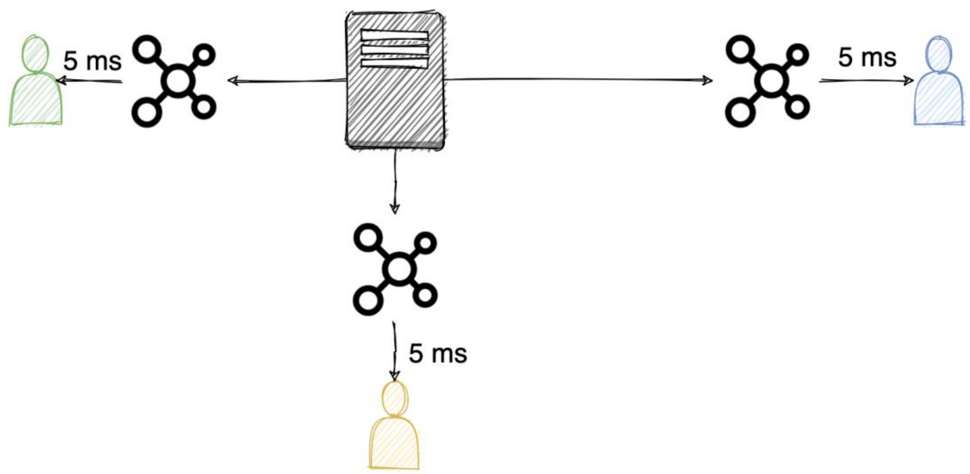
据我了解，某东在全国热门城市建立仓库，每个仓库都有热门产品的储备。当用户下单后，通常会选择离用户最近的仓库发货。



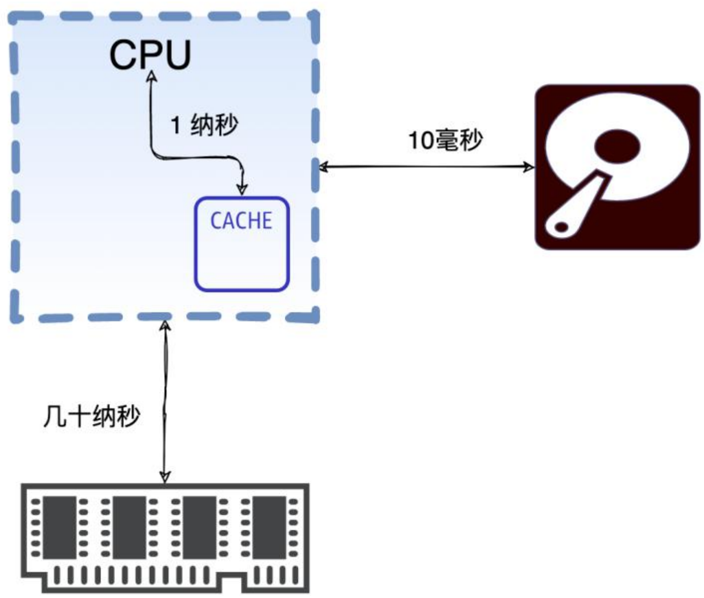
和上面的案例类似，如下图所示， CDN 加速也采用类似的机制。



比如之前只有一个节点，较远的用户访问耗时为 1 秒钟，通过 CSN 加速之后，较远的用户也可以有较好的响应速度。

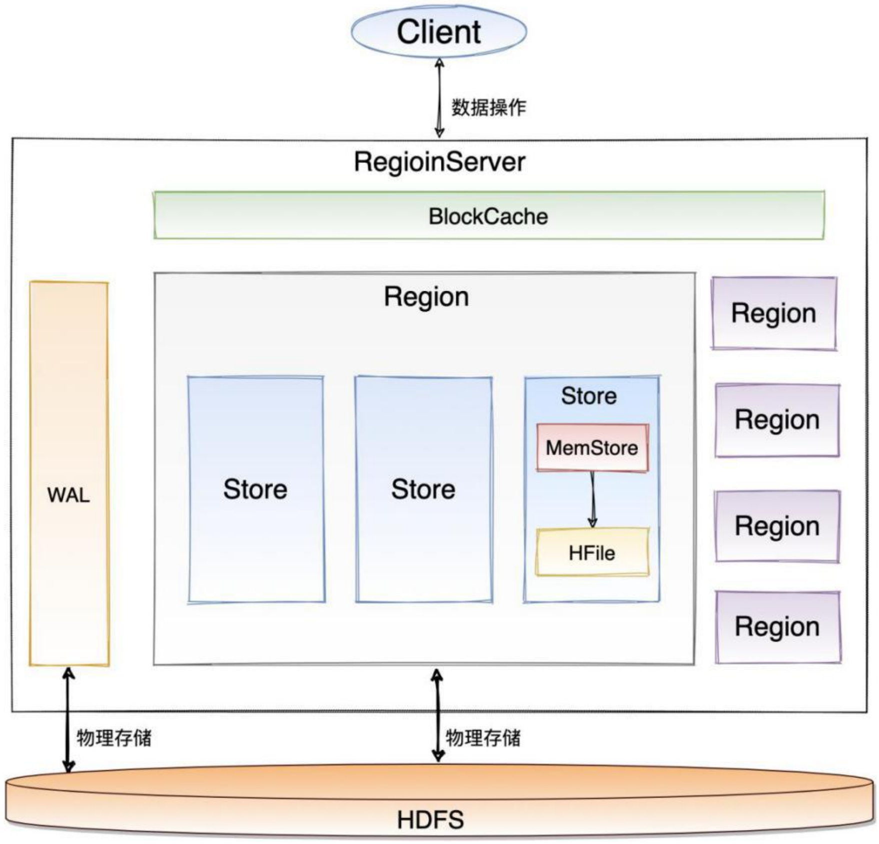


前面给出了不同系统组件的操作时差图，下面给出一张 CPU 缓存、内存和磁盘操作的时差参考图[4]。

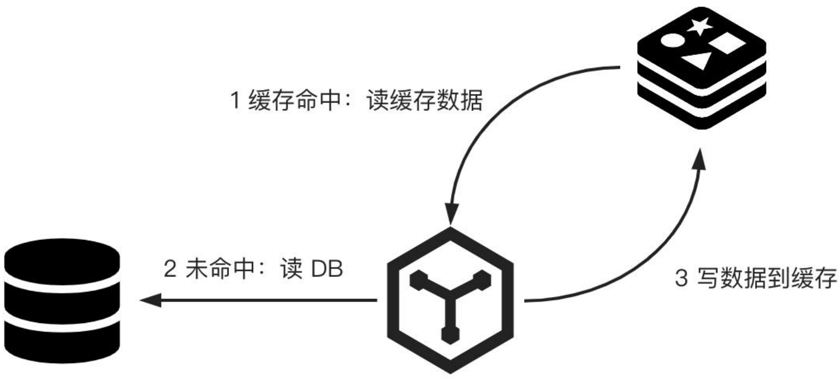


鉴于内存的读写速度比磁盘快很多，通常设计方案时会选择将热点数据缓存在内存中，这样就可以加快访问速度。

HBase 会在 RegionServer 中包含 BlockCache ，当读请求到 HBase 后，会先尝试查询 BlockCache， 如果获取不到再去 HFile 和 MemStore 中获取，如果获取到了该数据，则返回给上游的同时将 Block 块缓存到 BlockCache 中[6]。



实际开发中最常用的是类似 Redis 的 KV 缓存或如 Guava 内存缓存。基本流程是先读缓存 ，如果未命中则读数据库，将数据缓存到缓存中。



在实际开发过程中，如果整个业务流程中需要多次调用同一个接口，可以采用线程级别（包括在同一个线程中，也包括在父子线程）缓存，避免在同一个流程中对同一个接口相同参数重复发起请求。

如使用模板模式设计某个业务流程时，子步骤中都需要对同一个订单号的订单内容进行查询或者对同一个课程信息进行查询。

此时，可以直接使用 java.lang.ThreadLocal 或者 java.lang.InheritableThreadLocal （可以完成父线程到子线程的值传递），也可以使用 com.alibaba.ttl.TransmittableThreadLocal （使用线程池等会池化复用线程的执行组件情况下，提供 ThreadLocal 值的传递功能，解决异步执行时上下文传递的问题。具体用法请参考其官方文档）。

参考代码示例如下：

```java
public class SomeContext {

private static final ThreadLocal<Map<String, Object>> CONTEXT = new TransmittableThreadLocal<>();

/**
 * 初始化上下文
 */
public static void initContext() {
    Map<String, Object> content = CONTEXT.get();
    if (content == null) {
        CONTEXT.set(new HashMap<>(16));
    } else {
        CONTEXT.get().clear();
    }
}

/**
 * 清除上下文
 */
public static void clearContext() {
    CONTEXT.remove();
}

/**
 * 获取上下文内容
 */
public static <T> T getValue(String key) {
    Map<String, Object> content = CONTEXT.get();
    if (content == null) {
        return null;
    }
    return (T) content.get(key);
}

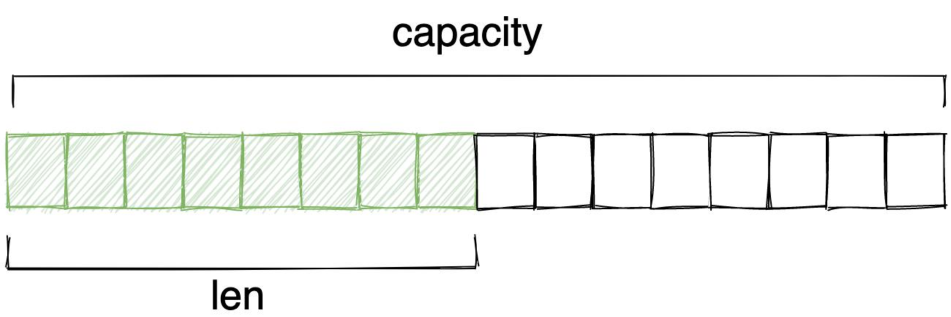
/**
 * 设置上下文内容参数
 */
public static void putValue(String key, Object value) {
    Map<String, Object> content = CONTEXT.get();
    if (content == null) {
        CONTEXT.set(new HashMap<>(16));
        content = CONTEXT.get();
    }
    content.put(key, value);
}
}
```

另外一种空间换时间的方式是预留（侧重于空间维度）。

比如 ArrayList 是一种基于底层基于数组的”变成数组”，当数据超过能够存储的长度时，并不会增加一个元素就扩容一次，而是每次扩容到原来数组长度的 1.5 倍。

```java
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

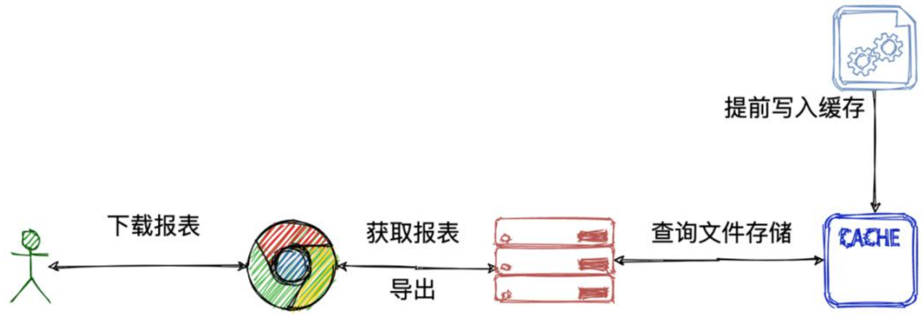
Redis 中字符串也采用预分配冗余的存储空间的方式减少内存的频繁分配。如下图所示，当字符串分配的实际空间为 capacity 一般要高于实际的字符串长度 length。当字符串占用的存储空间小于 1M 时，扩容为当前空间的一倍；当字符串占用的存储空间大于 1M 时，扩容只扩容 1M ，但是最大长度为 512M[3]。



还有一种空间换时间的方式为**预先处理/预热/ 静态化（侧重时间维度）**。

可以将一些不容易变动且瞬间访问量可能较大的数据，可以预先加载到缓存中。可以将不变动的热点页面静态化，放到 CDN 节点中加速访问。

比如在某个确定的环节，报表内容就已经可以确定下来，在用户需要下载报表之前提前将报表放在缓存或者专门的文件加速服务器中，这样用户来的时候就可以快速下载。



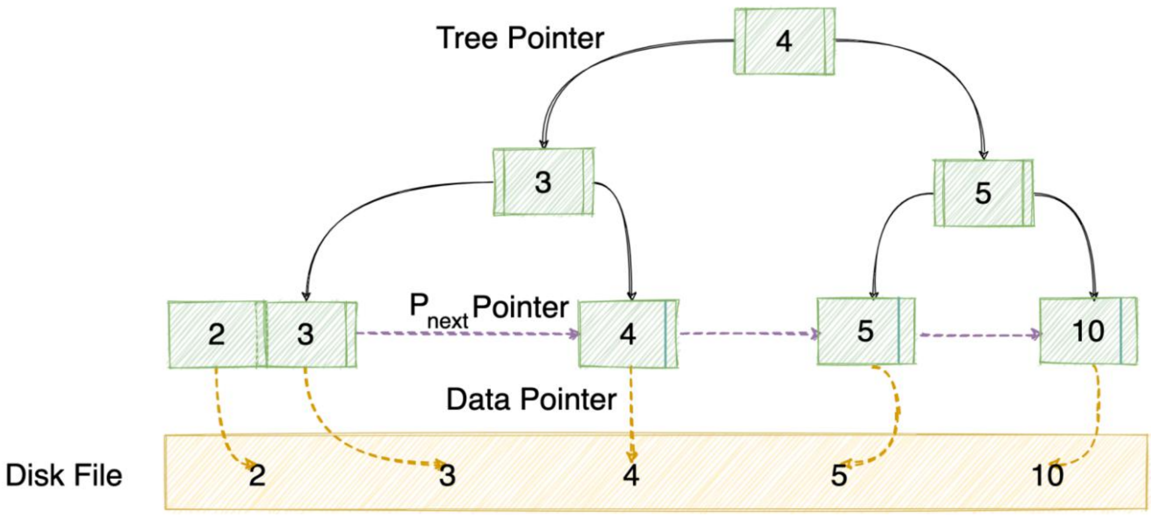
其实上面的某东快速送货的案例也体现出了预留的思想。一般来说，热门商品不可能用户买的时候再去生产厂家订做，都会提前在仓库中储备一定量的货物，用户下单时直接发货即可。

其实**索引也是典型的空间换时间的做法**。

这里的索引包括文件系统索引、数据库系统索引等。他们的核心思想都是一样的，先为要检索的内容建好索引，在查找时根据索引快速寻找对应的数据。

我们在开发时，通常会用到 MySQL 、Oracle 数据库，为了提高查询速度，通常会设计索引，让索引能够覆盖我们的查询条件。

以 MySQL 的 InnoDB 引擎来说，使用 B+ 树的结构为索引字段建好索引。在查询时如果命中索引，可以通过索引快速找到对应的数据。

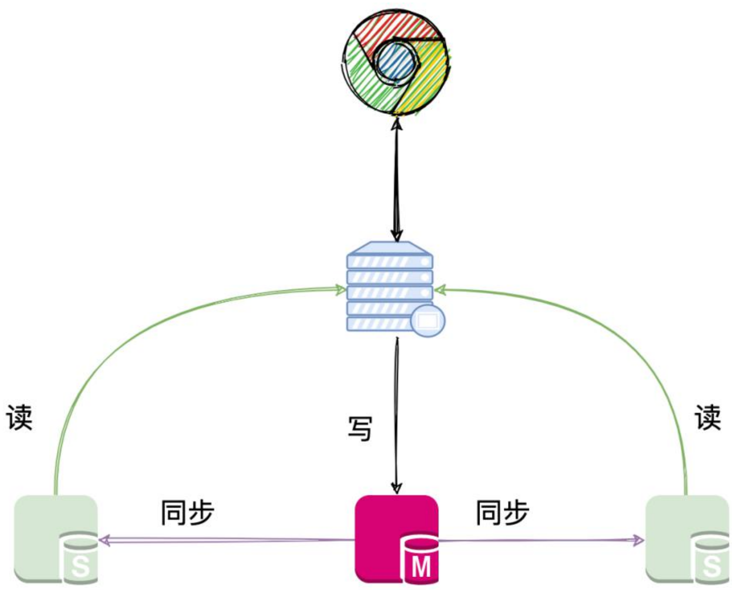


很多架构的设计都是用空间换时间的思想实现性能优化的，如集群架构、读写分离、分库分表、分布式架构。

由于单机承载量的有限性，可以通过加机器化整为零，分担请求。

然而，机器之间不同的组织方式形成了不同的架构模式。

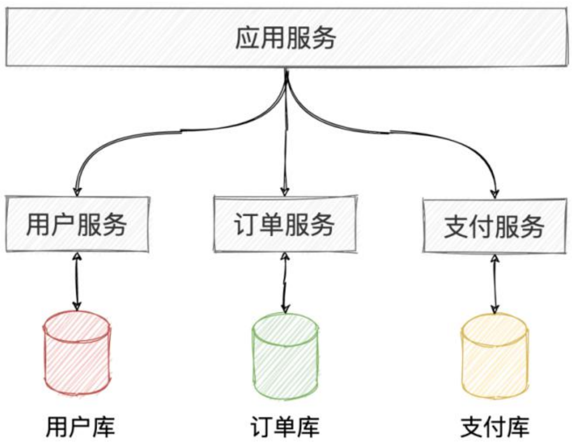
通过 nginx 负载均衡将请求分发到不同的实例上，称为集群模式。通过写操作写主实例，读操作读从实例，通过主从复制读写分离的方式，提高了整体吞吐量。



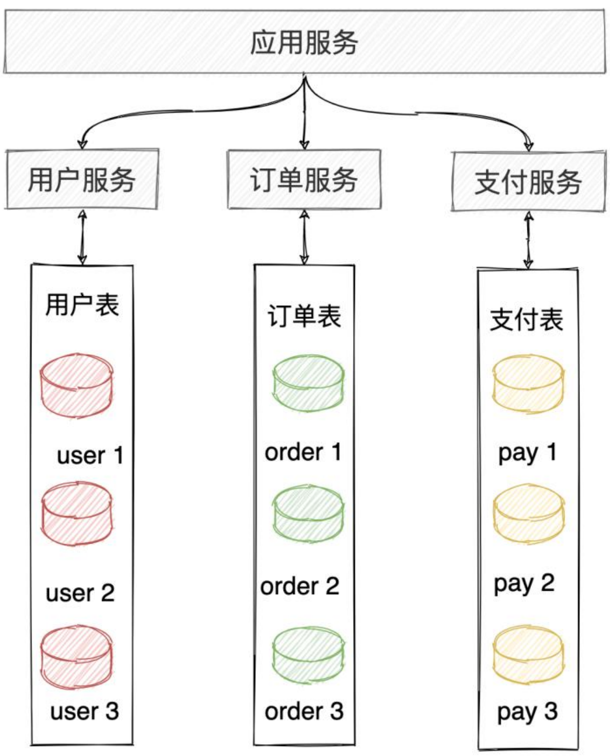
以上两种模式的共同点是每个节点都拥有相同的数据。

在某种条件下，需要将存放在一个数据库中的数据分散存储到多个数据库（主机）上，达到分散单机设备负载的效果。就涉及到了数据的切分，主要包括水平拆分和垂直拆分。

垂直切分，通过将不同业务的数据，放到不同的数据库中，实现不同业务之间数据库层面的隔离。



水平切分可以按照某种规则将某些字段分散到多个库中，每个表中只包含一部分数据[5]。

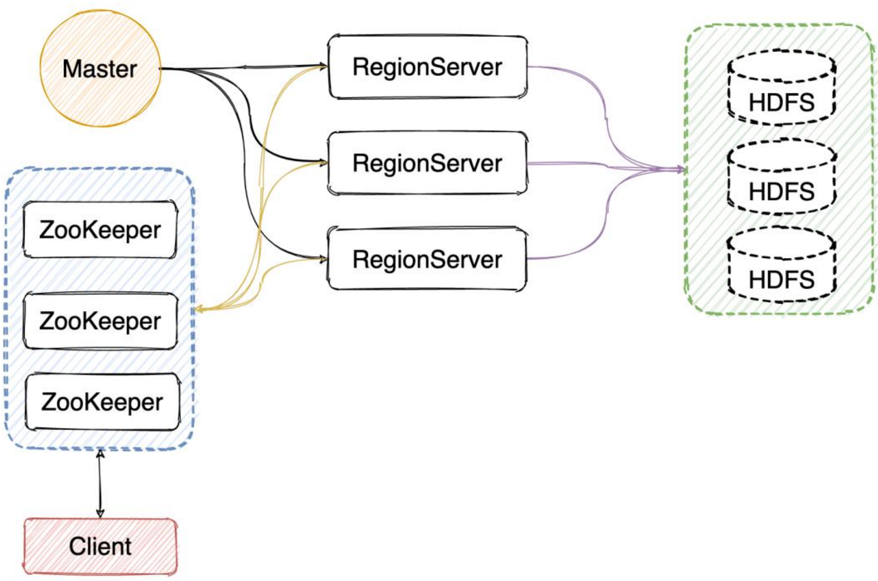


随着互联网的不断发展，访问量、数据量不断增多。为了突破单机的 CPU 处理能力、内存、磁盘的限制，很多**分布式中间件**应运而生。

分布式系统是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统。分布式系统的出现是为了用廉价的、普通的机器完成单个计算机无法完成的计算、存储任务。其目的是利用更多的机器，处理更多的数据。

HBase 的架构就鲜明地体现着分布式的思想。

下图为 HBase 的整体架构。



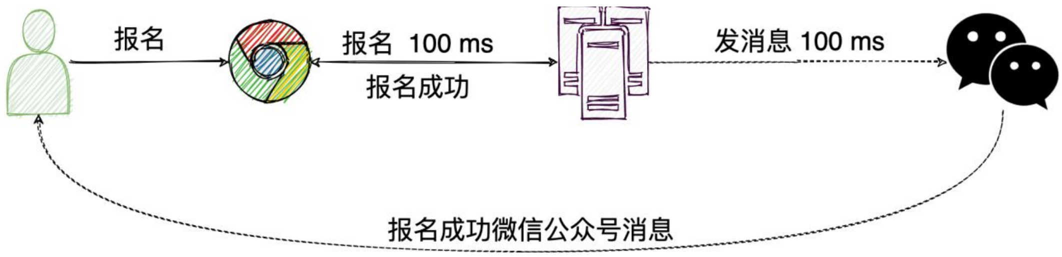
Master 负责各种协调工作，ZooKeeper 扮演着管家的角色，管理 HBase 中所有 RegionServer 中的信息。Region 用来存放数据， RegionServer 是 存放 Region 的容器。最终可以通过 HDFS 持久化数据[6]。通过增加 RegionServer 和 HDFS 机器就可以增加整体的承载能力。

4.3.2 同步转异步

通常比较耗时或者不必放在主流程中执行的任务，可以考虑使用异步的方式来处理。

可以使用线程池或者结合 CountDownLatch 、CyclicBarrier 和 CompletableFuture 等 API 并发执行。

比如用户在系统上报名，后端在处理完报名逻辑后，如果报名成功需要给用户推送公众号发送报名成功的消息，此时直接返回给用户成功即可。



发送公众号消息的任务可以放在线程池中异步执行。

假如后端逻辑需要执行 100 毫秒，发送公众号消息需要执行 100 毫秒，如果同步发送消息，耗时为 200 毫秒。如果发送公众号消息的环节采用异步的方式，那么对用户而言只需要等待 100 毫秒即可。

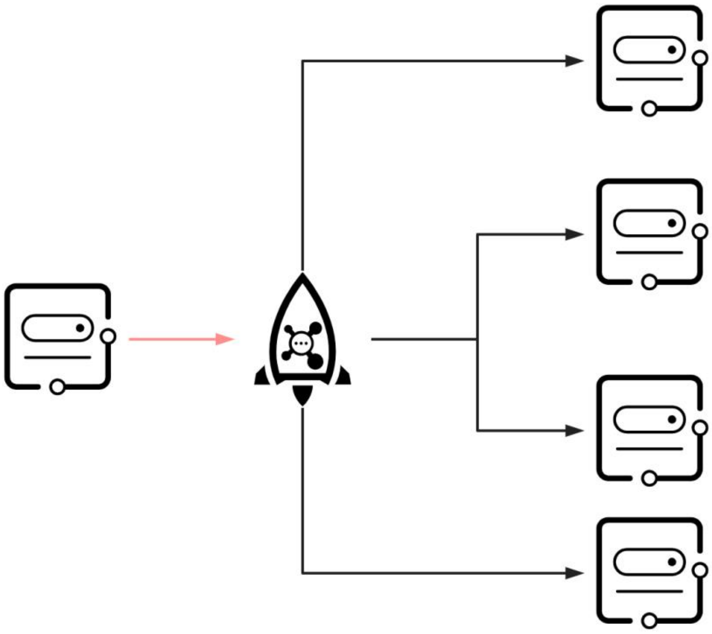
同步转异步还包括延迟加载。

比如在领域驱动设计时，某个聚合根中的实体并不是所有场景都需要用到，构造该实体需要查询其他表，此时可以采用延迟加载的方式，在获取该属性时再去查询构造该实体。

4.3.3 串行转并行

如果某个任务包含多个步骤，串行执行时间时每个步骤耗时相加才是最终的耗时，容易超时或者体验不好。

可以考虑将这些子任务使用 ForkJoinPool、 Stream 并行流操作。还可以将大任务拆分成多个小任务，通过消息队列或者大数据框架并行执行。



比如需要快速将 100 万条消息发送出去，可以每 100 条封装成一个子任务丢到 MQ 中，负载均衡到多个消费者服务器上，并行执行，而且每台机器还可以使用 10 个线程并发发送。

4.3.4 降低冲突的范围

常见的降低冲突范围的方法有：如偏向锁、分段加锁、读写锁、**CopyOnWrite**、使用乐观锁、隔离等。

为了实现没有多线程竞争情况下，减少传统重量级锁使用操作系统互斥量带来的性能损耗，JDK1.6 之后还引入了轻量级锁。

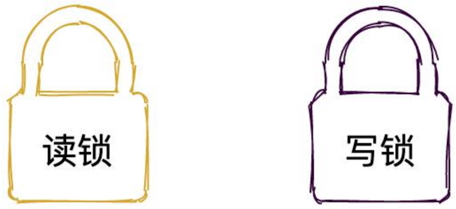
为了提高性能， JDK 1.6 引入了偏向锁，它偏向于第一个获取它的线程，如果在接下来的执行过程中，没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。

对于临界资源的访问，如果不得不加锁，要尽量降低锁的范围。

如采用分段加锁机制，不在同一段的数据就不会因为这一段内有写操作而相互影响。

如数据库中能锁行就不要锁表。读锁和读锁之间并不互斥，读锁和写锁、写锁和写锁才互斥。

数据的操作一般就分为两类，一种是读，一种是写。在读多写少的场景下，可以通过读写锁、乐观锁的方式降低冲突的范围或者说降低冲突带来的性能损耗来提高性能。



为了更好地帮助大家理解什么是降低锁的范围，下面举一个栗子：

比如某个用户只能对某个商品下一个订单，为了防止用户重发下单一般需要先查询是否有订单，如果已经有一个订单则不允许再次下单。

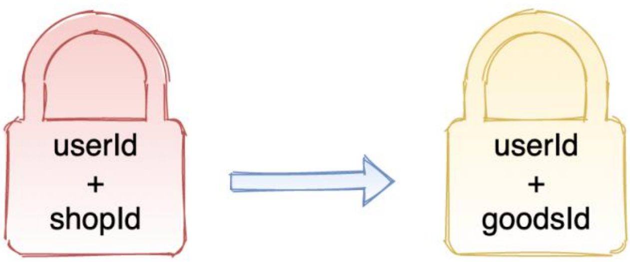
在高并发场景下，不加锁，同时有两个并发下单请求过来，都查询订单表，发现没有数据，都插入下单记录，这就尴尬。

可能有经验的同学已经有思路了，加锁呗！

有些同学可能就开始设计了 key：userId +"_"+shopId。（实际开发中可能很多人并不会这么傻，这里只是举个例子，帮助大家理解)

产品要求的是不能对同一个商品重复下单，没说是不能对同一个店铺的不同商品重复下单。
显然上面锁的范围扩大了。

应该将用户 id 和 商品 id 组合在一起构成分布式锁的 key： userId +"_"+goodsId。



4.3.5 空间局部性

先介绍下时间局部性和空间局部性的概念。

时间局部性：如果在某一点时访问了存储器的特定位置，则很可能在不久的将来将再次访问相同的位置。

空间局部性：如果特定存储位置在特定时间被访问，则很可能在不久的将来访问附近的存储位置。

其实时间局部性是加缓存的最主要依据。

那么我们如何利用空间局部性进行性能优化呢？

我们先看一下 MySQL 中的一个案例：

我们知道读写磁盘的速度非常慢，和内存读写差了几个数量级。

如果我们想从表中读取一些数据时， InnoDB 存储引擎会一条一条把记录从磁盘中读出来吗？

InnoDB 采取的方式是：将数据划分为若干个页，以页作为磁盘和内存之间交互的基本单位，InnoDB 中页的大小一般为 **16** KB。也就是在一般情况下，一次最少从磁盘中读取 16KB 的内容到内存中，一次最少把内存中的 16KB 内容刷新到磁盘中[7]。

其实 MySQL 的 InnoDB 存储引擎这么做的主要理论依据就是空间局部性，这点和合并操作非常类似。

4.4 其他

4.4.1 提前处理

我们未必要等到用到数据时才去计算数据，可以提前一些计算出来，这样用到的时候性能就更好。比如推荐服务的某个步骤需要调用推荐内容的一些特征，如果这些特征可以提前计算好，那么调用

时就可以节省大量的时间。

4.4.2 实时转离线

在实际业务开发中，实时计算出某个数据难以实现或者性能很差，如果产品能够允许的情况下，可以转为离线计算。

4.4.3 随机读写转顺序读写

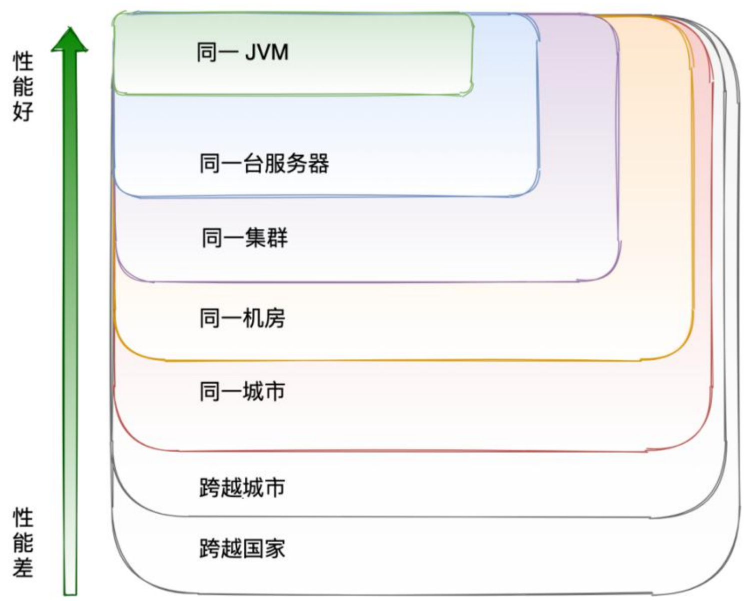
随机 IO 读写速度和顺序 IO 读写速度差距较大。
因此有可能的话，尽量将随机读写转为顺序读写。

4.4.4 就近原则

前面讲到使用缓存达到空间换时间的效果。

但有时候还可以采用 “就近原则” ，使用”更近”的数据，调用”更近”的服务。

一般来说，同一个虚拟机 > 同一台服务器 > 同一个集群 > 同一个机房 > 同一个城市 > 同国其他城市 > 跨国 （> 表示优于）。



Dubbo 从 2.2.0 开始，每个服务默认都会在本地暴露。在引用服务的时候，默认优先引用本地服务。如果希望引用远程服务可以使用一下配置强制引用远程服务[8]。 本质上也是采用“就近原则”，优先调用本地服务，从而减少不必要的网络耗时。

4.4.5 选择合适的数据结构和算法

不同数据结构有各自的优势和劣势，有各自的适用场景。

如去重，尤其是数据量较大时，可以利用 Set 的元素唯一性的特性，快速对一个集合进行去重操作，避免使用 List 的 contains 方法进行遍历、对比、去重。

有些场景下可以使用 BitSet 、Bag 等数据结构性能会更好。

再比如 HashMap 在哈希冲突时先拉链，冲突超过 8 个转为红黑树。当冲突较严重时，红黑树的性能显然比链表更高，这其实就是根据情况结合不同的数据结构的优势实现性能优化的典型案例。

大家都知道不同算法的时间复杂度不同，不同算法的耗时有巨大的差异，在实际开发中涉及到算法部分一定要注意时间复杂度问题。

在工作中就遇到过有人用 List 的 contains 方法去重，当数据量较大时耗时会很久。

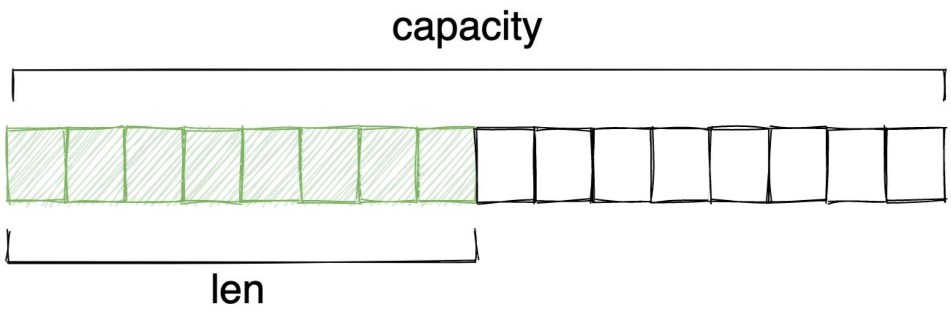
有些场景下可以选择使用布隆过滤器等算法优化性能。

4.4.6 加限制条件（技术层面）

此外，添加一些限制条件也是性能优化的重要思想。

以 Redis 的动态字符串为例。

其内部实现思想和 Java 中的 ArrayList 非常相似，采用预先分配冗余空间的方式来减少内存的频繁分配。



不同的是，当字符串小于 1 M 时，扩容一倍；如果超过 1M ，每次扩容 1M 的空间，最大长度为 512 M。

这里的最大长度限制就是为了保证不因单个 key 不至于过大，导致慢查询。

4.4.7 CPU 密集型和 IO 密集型

对于 CPU 密集型任务，可以通过增加机器和提高机器配置，使用大数据框架，升级架构等方式来解决。

对于 IO 密集型操作，可以侧重考虑通过并行、异步、合并、预处理等方法进行性能优化。

4.4.8 根据技术特点去优化

具体到某个技术都有会辅助性能优化的命令或工具，需要大家自己去掌握。

不同的工具性能优化也有自己的特点，如 MySQL 设计好索引、HBase 设计好 rowkey 规则等，需要针对性地学习。

如 MySQL 可以使用 explain 指令分析语句性能进行分析；MySQL 创建联合索引时，将差异性最大的字段放在前面；比如业务上需要根据手机号后 5 位进行查询，除了设计表字段存储完整手机号外，可以额外设置一个字段仅存储后 5 位并加索引，这样就可以实现利用该索引提高查询速度；比如业务上需要将 MySQL 存储的大数据量导出时，避免使用 limit ，可以通过每次查询修改 where 条件来加快查询速度。如： SELECT * FROM sometable WHERE id>#{lastId} ORDER BY id ASC LIMIT 20;

如 HBase rowkey 的设计可以参考阿里云数据库 HBase 开发指南中的一些建议。

还有很多技术需要通过调整参数来优化，这些也需要通过官方文档、相关的权威图书、相关的技术专栏等针对性优化。

4.4.9 全链路优化

很多时候性能优化不能只把眼光局限于当前系统，需要考虑全链路，考虑前端和后端，当前系统和下游系统等。

有些性能优化问题，需要前端和后端通力合作，一起改进。

比如前端在某个页面执行了一些没有必要的调用；前端加载的资源速度过慢，可以考虑合并资源或者使用 CDN 加速等。

有些性能优化问题，需要后端和其下游通力合作，一起改进。

如果下游只提供某个功能的单个接口，而我们有批量执行的诉求，可能需要推动下游提供批量接口。如果下游接口有性能问题，可能还需要推动下游进行优化。

4.4.10 多种手段相结合

不同的数据结构和算法、不同中间件、不同的框架和架构，都有最适合的使用场景，都有各自的优势和劣势。在实践中，往往需要多种性能优化思想结合在一起来解决问题。

如某个业务将 MySQL 、 Redis 和 Es 多种存储方式结合一起使用，发挥各自的优势；如从技术层面和产品层面相结合进行优化。

在实际进行性能优化时要结合具体的场景，进行权衡之后做出选择。

比如有些数据可以存到 MySQL 表中，有些热点数据可以存到缓存中；快照数据可以存储到 HBase 中；文件存储可以使用专业的文件存储服务；需要进行模糊搜索的条件可以放到搜索引擎中。

比如在设计某个功能时，需要多表联查，此时可以通过在一个表中冗余另外一个表的字段来避免联查操作；还可以通过 ES 等技术做一个大宽表来避免 DB 层多表联查，来提高性能。

比如将单库改成读写分离架构，或者改成分库分表架构，将单体应用拆分成多个微服务，甚至采用单元化架构等。

4.5 产品层面

不是所有的性能优化都是要通过技术手段来实现，如果技术手段不容易解决，可以考虑从产品设计层面来实现。

4.5.1 加限制条件（产品层面）

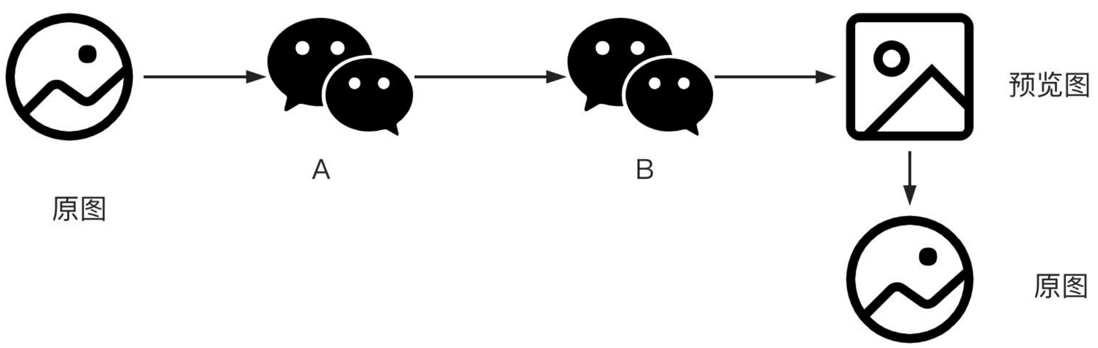
如果从技术层面性能优化不好实现，可以和产品经历协商，让产品经历做出一些妥协，加上一些限制条件，很多问题就迎刃而解。

比如产品层面可以**限制用户上传的图片、文件大小，限制录入的文本大小等**。

比如提供不同的版本，默认选择性能较好的版本。如视频网站提供不同的清晰度，默认展示普通清晰度的视频或者根据用户网速选择最优的清晰度。

还可以和产品同学进行交流，砍掉某些不合理的搜索条件等来获得更好的性能体验。

如微信发送图片时，即使发送时你选择原图，对方接收到的一般都是预览图，只有打开后选择”下载原图” 才会真正去下载原图到本地。



此外，微信对视频或其他类型的文件传输也有大小的限制。

这些都是通过加限制条件来辅助性能优化在产品层面的体现。

限制搜索数据翻页的页数

再比如，某个业务的数据采用 ES 索引，为了避免深翻页问题，产品层面限制用户只可以查看前 200 页数据，支持按照关键的几个字段排序。

提供前 200 页数据的查询已经可以满足业务需求，如果真需要查看所有数据，提供数据导出的功能。

4.5.2 砍需求

优秀的开发也要有一些产品思维，要懂得站在用户角度了解真正的诉求，懂得权衡某些设计的利弊，权衡投入产出比等。

可能下面这句话不是很正能量，但的确是解决问题的一个思路。



当产品提出一些不太合理的需求、提出的需求业务价值不高、投入产出比不高时，从技术层面不好解决时，可以考虑说服产品砍掉需求。

五、注意事项

5.1 避免过早优化

在设计性能优化方案时，要注意避免过早优化，要考虑投入产出比。

在产品初期，只要性能不会特别影响用户体验，通常不需要特别关注性能问题。

可以把更多的精力投放到满足用户的核心需求，提高产品销量，让产品生存下来。

5.2 考虑其他指标

在考虑性能优化的同时要注意可读性、可用性、稳定性、正确性、可拓展性、安全性等。

一般来说，软件设计的原则应该优先于性能优化的原则。即不能为了性能优化而破坏软件设计的原则，给后续的拓展维护埋坑。比如设计要符合 “高内聚、弱耦合”、“封装复杂度”；比如设计模式的一些基本原则：“单一职责原则”、“迪米特法则”、“开闭原则”、“里氏替换原则”、“接口隔离原则”、“组合复用原则” 等。

在数据量不是超大的情况下，纯内存操作相对是比较快的，千万不要为了省一两次 for 循环而写出非常难以读懂的代码。

如在使用缓存来加快访问速度时，要考虑必要性。还要考虑数据库和缓存的一致性问题。

在实际工作中就遇到有同学为了提高性能，对某个看起来一定不会修改的数据加了缓存，超时时间设置好几十分钟，然而由于忽略了某个导致变更的入口，最终因缓存和实际数据不一致引发线上故障。这里有个小技巧，加缓存时一定要慎重思考缓存超期时间，如果没有太大把握，可以加个动态开关在出现问题时可以及时调整。如果没有太大把握，在能满足业务需求的情况下，尽量将超期时间设置短一些。

再比如， fastjson 虽然宣称性能很不错，但是屡屡爆出安全漏洞，给用户带来很多困扰，很多人也不会选择这个框架，这也说明性能不是全部。

5.3 优化体验

有些时候，在现有技术条件下无论怎样都很难优化到让人满意的效果时，可以想办法提升用户体验。比如很多视频平台在视频卡顿时会增加 loading 效果；很多银行 app 转账时，会有、“等待 xx s”的倒计时；支付宝的转账支持通过账单详情查看转账的处理进度。



比如我们设计某数据导出到 excel 文件供用户下载的功能，可以在用户下载前的某个时机提前生成好，也可以参考上面的性能优化方法提高生成的速度。但如果怎么提高速度都很难在一两秒钟处理完成，如果在相对能够接受的时间内，如 10s ，能处理完成可以前端加 loading 效果，如果还需要更久的时间，即使加上 loading 用户也很难接受，则可以参考上述做法，异步查看处理进度。

六、实际案例

6.1 案例描述

下面给出一个模拟的业务场景，大家可以结合上面给出的性能优化核心思路，自己先设计一个性能优化的方案再和给出的方案进行对比，如果自己设计的方案更好，为你点赞；如果设计的方案没有给出去的方案更好，对比下差异。

注：本文给出的方案未必是最佳，只是希望帮助大家理解性能优化思想如何应用。

功能描述：

用户浏览活动时根据是否订阅过该活动，来决定是否展示 “订阅提醒” 按钮。 用户可以通过点击“订阅提醒”进行订阅，用户订阅后活动开始前 10 分钟发送微信公众号消息提醒用户参与活动。用户量可能较大，预计每个活动最大可能有几十万甚至上百万。 同一个活动可以开展多轮，下一轮订阅的时候只发送当前轮次订阅的这一批人。

需要思考的问题：

- 数据存在哪里?
- 如何尽快把消息发出去?
- 大量的消息发送会不会超过下游承载能力?
- 每次浏览活动时都要查询当前用户是否订阅过该活动，如果瞬间涌入大量流量怎么办?
- 如何保证查询的性能?
- 大量发送完毕的订阅数据如何清除?
- …

6.2 设计分析

问题 1：数据存在哪里?

单个活动就可能上百万的数据量，如果使用 MySQL 存储就要考虑分库分表，外加读写分离。

如果存储到 Redis，会造成大 key，就需要采用化整为零的思想，采用类似多级索引的方式，将订阅数据分拆到多级 key 中，实现起来很麻烦。

如果使用 MySQL 表来存储当前订阅的轮数，代价有点大。可以使用 Redis 存储某个活动当前订阅的轮数，发送完提醒后轮数 +1。

订阅数据可以使用 HBase 进行存储，将 活动 id（activityId），用户 id（userId），批次 id（batchId） 作为 rowkey 。

为了尽可能消除热点的影响，并加快 scan 速度，将每个活动的 rowkey 再拆分到 16 个桶中。如将用户 ID %16 得到 0 到 15 称为 bucketId，最终 rowkey 结构类似如下：

```
bucketId = userId % 16
prepare = bucketId +'_'+ activityId +'_'+  userId +'_'+ batchId
rowkey = sub(md5(prepare),0,5)+ '_'+ prepare
```

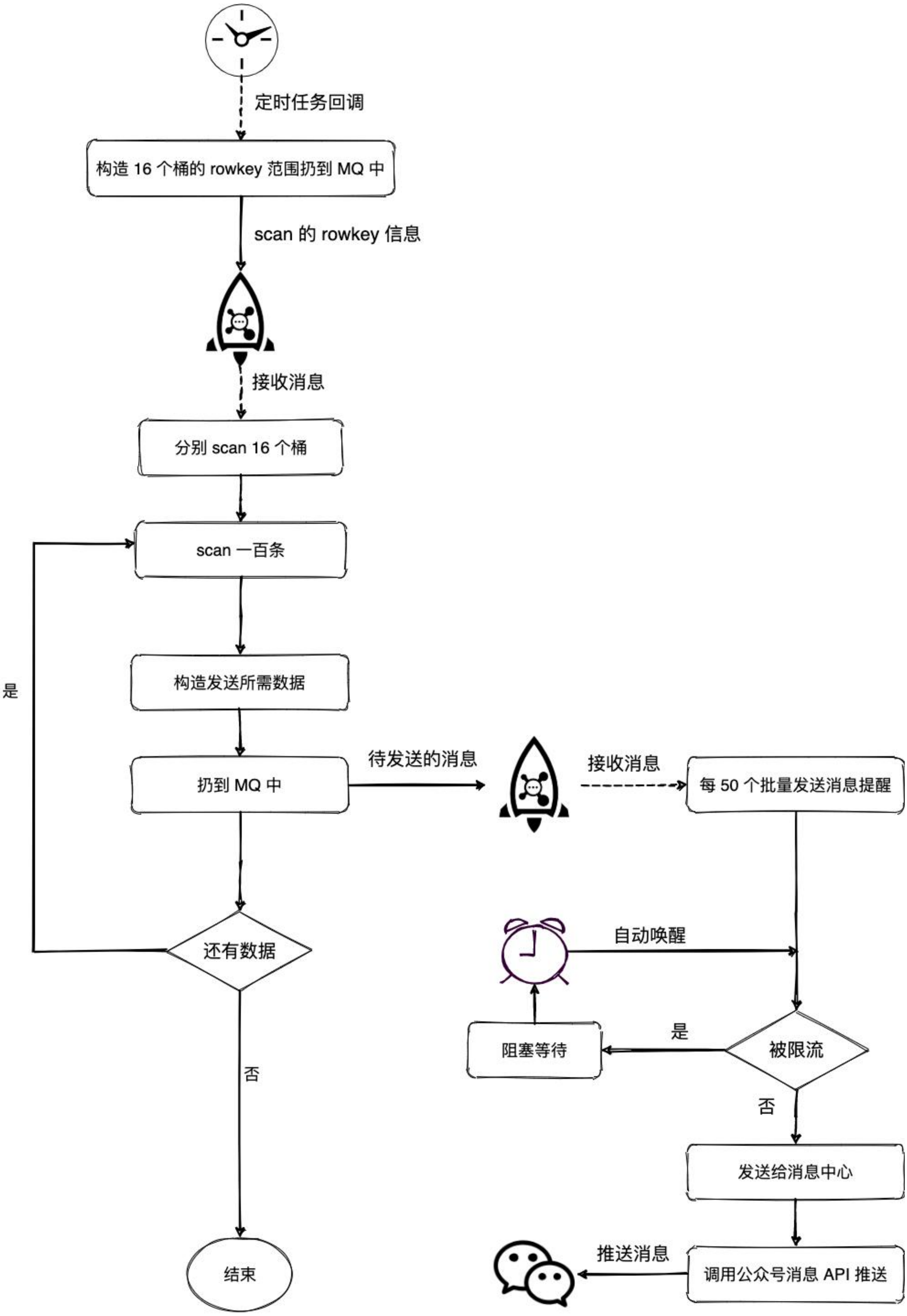
HBase 开启布隆过滤器，在判断当前用户是否订阅时，如果没有订阅，可以快速得到结果。

同时为了灵活性，桶的数量是通过 Apollo 来读取，如果有性能问题可以动态增加桶。

问题 2：如何保证消息尽快发出去？

我们使用定时消息，活动开启前 10 分钟回调过来，执行发送逻辑。

拼接 16 个桶的 scan 所需的 rowkey_start 和 rowkey_end ，发送到消息队列中，通过消息队列分发到多台机器并行 scan 每一个桶的数据。


定时任务回调
构造 16 个桶的 rowkey 范围扔到 MQ 中
scan 的 rowkey 信息
接收消息
分别 scan 16 个桶
scan 一百条
构造发送所需数据
扔到 MQ 中
待发送的消息
接收消息
每 50 个批量发送消息提醒
是
还有数据
否
结束
自动唤醒
是
阻塞等待
被限流
否
发送给消息中心
推送消息
调用公众号消息 API 推送

每次 scan 100 条扔到 MQ 中等待下游消费。

下游接收到消息后，每 50 个为一批发送给消息中心（消息中心负责消息的统一封装，并且提供了批量接口并限制一次 RPC 调用最多参数中传入 50 个消息）。

为了避免将消息中机器打爆，对消息的发送做了限流控制，将速率控制在每秒 6000 QPS 以内。

问题 3：过期如何清除？

通过调研，发现公司的 HBase 架构目前还存在一些问题，调用 API 删除数据时不能确保百分比清除。

最终和产品交流加了限制条件，确定活动可以选的最长提醒时间为 1 年，HBase 设置数据自动清空时间为 1 年。这样就不需要再重新 scan 调用 API 去清空数据，同时也减少了资源占用。

6.3 案例总结

在上述的设计案例中，用到了化整为零、同步转异步、合并操作 的优化思想；用到了根据存储方式（MySQL、Redis、HBase）的特点和算法（布隆过滤器）进行性能优化的思路。

设计中也有一些 加限制条件的地方：

- 技术方面：下游的消息中心对上游有 QPS 限制，对批量接口的大小也做了限制。
- 产品方面：为了配合技术方案，产品对发送消息的有效期做了限制，并且不支持用户取消订阅。

整个方案都是在不断权衡中确定下来的。还有一些其他细节上的考虑，在这里就不详细论述。

希望大家可以通过这个案例，能够实实在在地体会到性能优化思想在实际项目中的应用。能够在未来的实践中，更好地利用性能优化方法论来解决问题。

七、总结

本书主要讲述性能优化的本质，性能优化的思想来源，性能优化的常见思路，还讲到了性能优化的注意事项。最后结合一个具体的场景讲述性能优化如何落地。

性能优化没有标准答案，优秀的程序员应该灵活将多种优化方案结合一起灵活运用，根据实际的场景做出权衡。

另外，希望大家能够意识到，本专题所讲的性能优化方面的知识并非只是帮助我们去做技术方案，还建议大家用性能优化的视角来学习中间件、源码，你将收获更多。

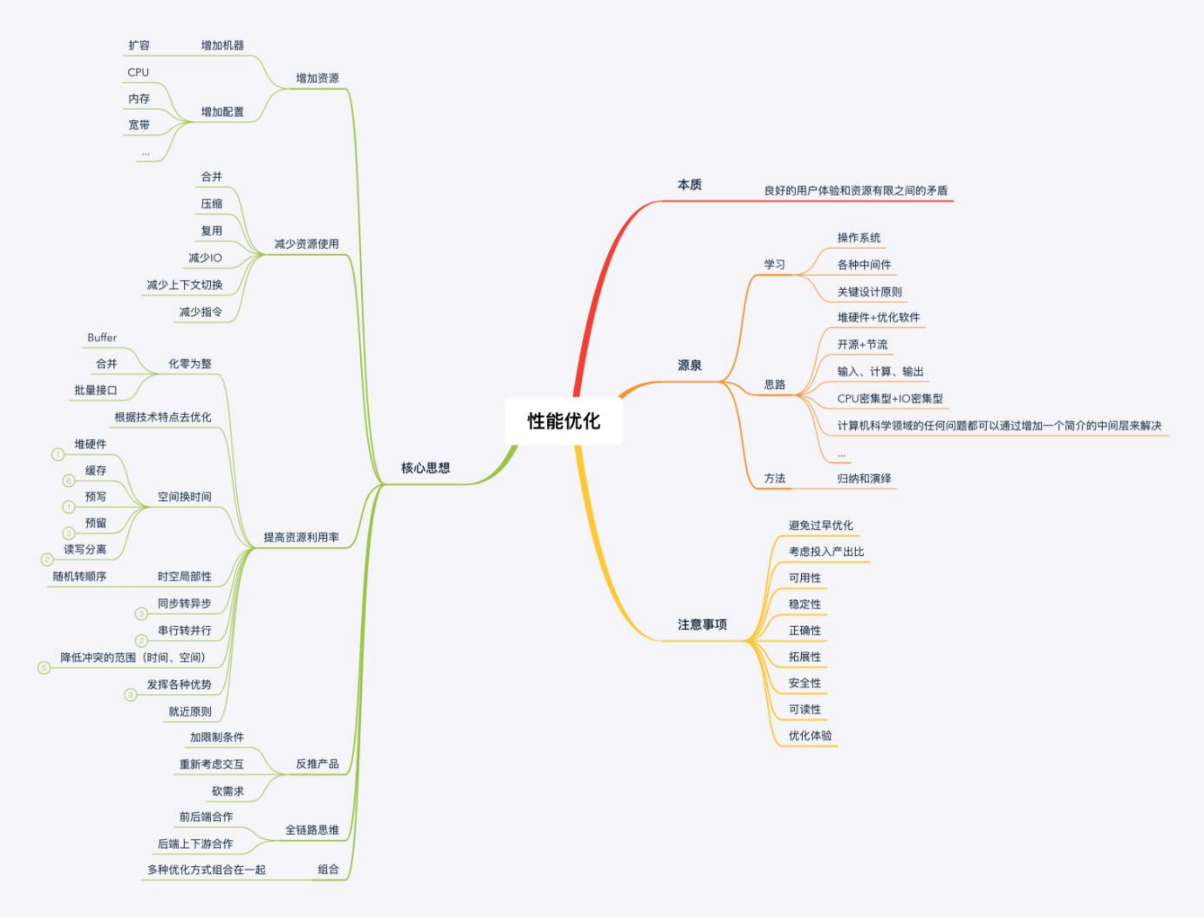
在这里推荐几本和性能优化相关的图书，大家感兴趣可以读读：《性能之巅》、《图解性能优化》、《Web 性能权威指南》、《Java 性能权威指南》、《深入理解计算机系统》等。

本书主要根据个人的技术经验编写，难免会有一些纰漏，如有错误欢迎批评指正。

最后，我想说："热爱可抵漫长岁月"。

希望大家能够在技术成长的道路上"Stay hungry，Stay foolish"，并且保持热爱，不断精进，一起加油!

附本书的思维导图：



可以通过以下方式获取配套资源（含思维导图）：《性能优化方法论配套资源》的原始文件。

参考资料

[1] 博文视点.《性能之巅：洞悉系统、企业与云计算》[M]. 电子工业出版社. 2015

[2] 林晓斌.《MySQL 45 讲》[M]

[3] 钱文品.《Redis 深度历险》[M]. 电子工业出版社. 2019

[4] [日] 小田圭二/ [日] 榑松谷仁 / [日] 平山毅/ [日] 冈田宪昌.《图解性能优化》. 人民邮电出版社. 2017

[5] Mycat 开源项目组. 《Mycat 权威指南》[M]

[6] 杨曦. 《HBase 不睡觉书》[M]. 清华大学出版社. 2018

[7] 小孩子 4919.《MySQL 是怎样运行的：从根儿上理解 MySQL》[M]. 人民邮电出版社. 2020

[8] Apache Dubbo 社区. 《本地调用》[M]

[9] Abel Avran & Floyd Marinescu[著] 孙向辉. 霍泰稳. 李琨 [译]. 《领域驱动设计精简版》。InfoQ 企业软件开发丛书[M]